New Contig Creation Algorithm for the *de novo* DNA Assembly Problem

Mohammad Goodarzi

Computer Science

A thesis submitted in partial fulfilment
of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science, Brock University
St. Catharines, Ontario

Abstract

DNA assembly is among the most fundamental and difficult problems in bioinformatics. Near optimal assembly solutions are available for bacterial and small genomes, however assembling large and complex genomes especially the *human genome* using Next-Generation-Sequencing (NGS) technologies is shown to be very difficult because of the highly repetitive and complex nature of the human genome, short read lengths, uneven data coverage and tools that are not specifically built for human genomes. Moreover, many algorithms are not even scalable to human genome datasets containing hundreds of millions of short reads. The DNA assembly problem is usually divided into several sub-problems including DNA data error detection and correction, contig creation, scaffolding and contigs orientation; each can be seen as a distinct research area. This thesis specifically focuses on creating contigs from the short reads and combining them with outputs from other tools in order to obtain better results. Three different assemblers including SOAPdenovo [Li09], Velvet [ZB08] and Meraculous [CHS+11] are selected for comparative purposes in this thesis. Obtained results show that this thesis' work produces comparable results to other assemblers and combining our contigs to outputs from other tools, produces the best results outperforming all other investigated assemblers.

Acknowledgement

I would like to thank my supervisor Dr. Sheridan Houghten who has made available her support in many ways which helped me achieving my research goals. I would also like to appreciate Dr. Ping Liang's guidances and suggestions through my research which helped me better understand the area and have access to his laboratory resources.

I am very grateful to have a family who have supported me in my way to continue my higher educations and finally, this thesis would not have been possible without the helps and sacrifices made by my beloved wife, Nazanin.

Contents

Chapter 1

Introduction to Genomes and Genome Assembly

1.1 DNA Molecule and Structure

Functions, activities and development of all living organisms are defined by a chemical molecule in their body called DNA. DNA is a macro molecule that consists of other simpler chemical units that encodes important genetic instructions defining how a living organism functions. Finding and analysing the sequence of chemical units in a DNA molecule is considered to be a key to understanding how living organisms work and finding cures for many genetic-related diseases. The importance of genetics and DNA analysis has created vast research areas in biology to find DNA structure and also in computer science to analyse massive amount of data generated in biology labs in order

to reveal important information about genetic codes. *Bioinformatics* is the general area of research that targets biology problems from the computer science point of view. This thesis focuses on solving one of the most fundamental problems in bioinformatics, the "*de novo* DNA assembly problem". Before going deep in to the main problem, an introduction about DNA structure, DNA sequencing technologies and genome assembly are presented in this chapter.

DNA consists of two long biopolymers made of simpler chemical units called *nucleotides*. These two long chains of nucleotides are connected to each other at every nucleotide location and can be imagined as a ladder. Each long chain is called a *strand*. There are four different nucleotides that are the basic blocks of the DNA molecule: Adenine, Cytosine, Guanine and Thymine which are abbreviated by the letters A, C, G and T respectively. Figure 1.1 shows the chemical structure of these nucleotides. Each pair of nucleotides in the DNA is called a *base*. Generally there is no preference for two bases to connect to each other in one strand but bases in equivalent locations in opposite strands must be complementary to each other. "A" is always complemented by "T" and "C" is always complemented by "G" and vice-versa [WC+53]. Figure 1.2 shows a very simple view of DNA molecule structure. For more detailed information about DNA molecule and its structure refer to [Nai07].

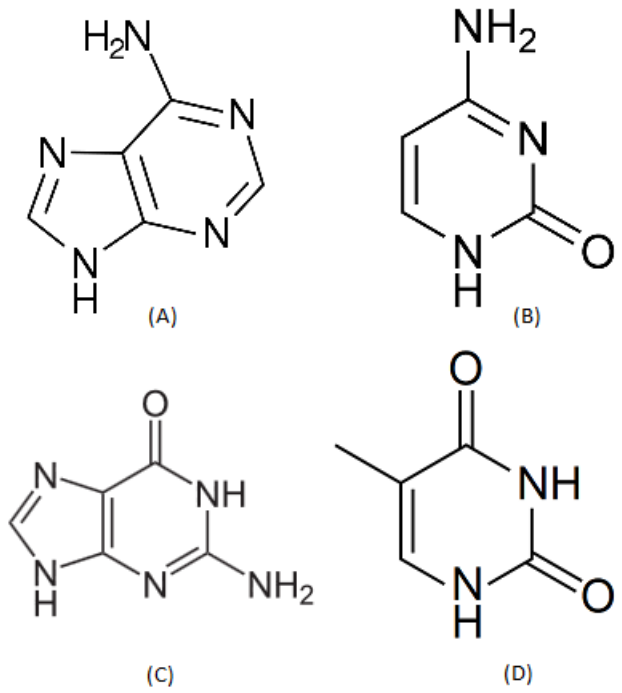

Figure 1.1: Chemical structure of nucleotides. (A): Adenine, (B): Cytosine, (C): Guanine, (D): Thymine. (Images source: http://en.wikipedia.org /wiki/Adenine, http://en.wikipedia.org /wiki/Cytosine, http://en.wikipedia.org /wiki/Guanine, http://en.wikipedia.org /wiki/Thymine)

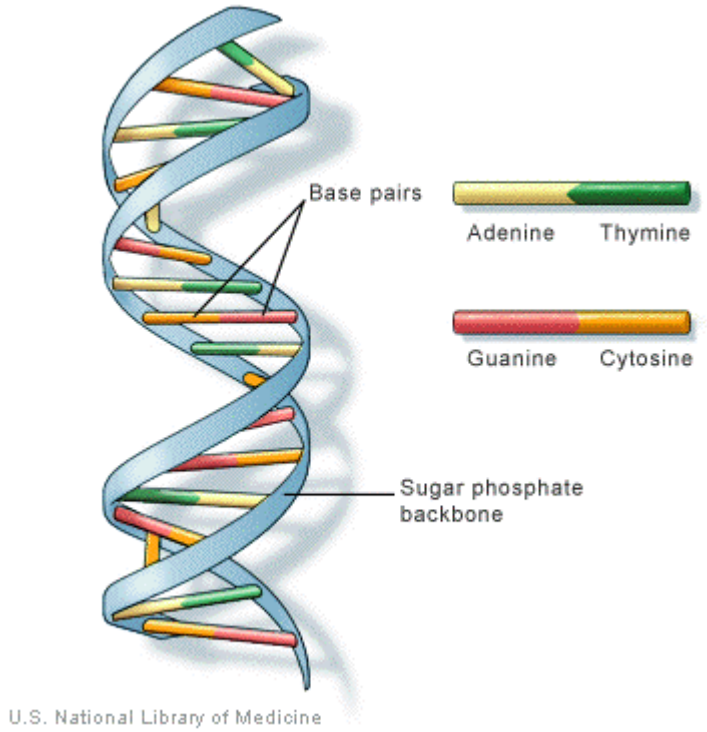

Figure 1.2: DNA structure (Image source: http://www.chemguide.co.uk/-organicprops/aminoacids/doublehelix.gif)

1.2 DNA Sequencing Technologies

Finding the sequence of base-pairs in a given DNA molecule is not an easy task. There has not been any approach to provide the complete sequence of DNA in a chromosome or a genome in a continuous form. This is mainly because DNA molecules are extremely large. For example, they consist of hundreds of millions of base-pairs in the case of mammalian genomes including the human genome. However, having knowledge of the DNA sequence of a genome is fundamental for other research areas in biology to progress. The first method to detect the precise order of base-pairs in a DNA molecule was devised by Fredrick Sanger in 1977 [SNC77] and this is still the most accurate method for DNA sequencing. Sanger-based sequencing technologies are able to extract base-pairs from fragments of the whole chromosomal DNA with a maximum length of around 1000 bp. DNA sequencing is an error prone process which may result in detecting wrong base-pairs from the DNA molecule. Two important problems around Sanger technology are its slow run time for large genomes and its cost. These limitations led to new technologies being devised addressing speed and price challenges. Next-Generation-Sequencing (NGS) technologies [DSC+10, MPC+09, HBB+08] were proposed from 1996 with the aim of reducing the cost and increasing the speed of the DNA sequencing process. From their time of invention until now, there have been numerous improvements in NGS technology and currently it is feasible to determine the DNA sequence of a genome comparatively quickly and cost effectively.

However NGS techrnologies also have several draw-backs:

- They produce even shorter sequence reads compared to Sanger sequencing. Currently the maximum length of DNA fragments produced by most NGS technologies is below 400 bps.
- They are more error prone than Sanger-based sequencing, especially in the starting and ending locations of fragments.

Illumina Genome Analyzer [DSC+10], Applied Biosystems SOLiD System [MPC+09], Helicos BioScience HeliScope [HBB+08], 454 Life Sciences [MEA+05] and Ion Torrent [RHR+11] are current leaders of Next-Generation-Sequencing technology.

Because it is not possible to sequence an entire DNA molecule in one attempt, researchers divide the large DNA molecule into chunks with lots of copies and perform the sequencing separately on every chunk in parallel, therefore obtaining sequences for all parts of the genome. The obtained sequences should be merged at the end to produce one continuous sequence of base-pairs for the base DNA molecule. *Shotgun Sequencing* [Pop04] is the technology that divides the DNA molecule into smaller parts in order to make the whole genome sequencing possible. Smaller DNA chunks produced by shotgun sequencing technology from random locations are called "Reads". Shotgun sequencing tries to produce random reads from all over the genome with even distribution, thus being able to produce the whole DNA sequence at the end. Figure 1.3 shows how reads are generated by shotgun sequencing

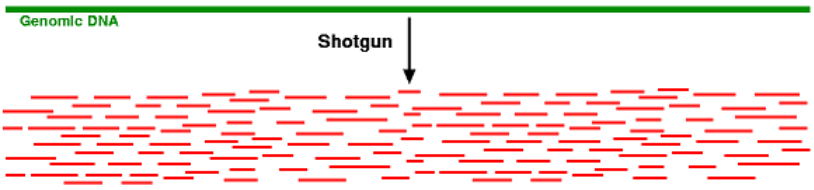

Figure 1.3: Shotgun sequencing. Small reads are created from random locations in the genome. Reads have overlap with each other making it possible to assemble them later, creating one contiguous sequence called "Assembly". (Image source: https://wiki.cebitec.uni-bielefeld.de/brf-software/images/2/2e/WholeGenomeShotgun.png)

for a genomic DNA.

Sequences obtained from random locations of the genome need to be processed in order to create one unique and continuous sequence expressing the base DNA sequence. Finding overlaps between the reads, merging the correct links together and expanding the reads to achieve larger sequences is the main task of "DNA Assembly" algorithms. This process is called *"de novo"* when there is not any other DNA sequence information about the species being sequenced. Sanger sequencing technology creates sufficiently high quality reads with enough length for DNA assembly algorithms to perform and extract the final assembly, but using NGS technologies imposes drastically different strategies in DNA sequence assembly. The DNA assembly problem can be solved if there are enough high quality reads from all over the genome that can resolve all complex repeating structures through the genome. Having larger reads helps to find better correct overlaps and lead to better results.

Coverage (read depth) is the average number of reads representing a given nucleotide in the genome. It can be calculated from the length of the original genome (G), the number of reads (N), and the average read length (L) as $(N * L)/G$ [MGG10].

Currently, shotgun sequencing is used along with NGS technologies to sequence new species with large genomes. This produces hundreds of millions of reads that need to be processed. Dealing with this huge amount of data needs careful considerations and algorithms, thus conventional DNA assembly algorithms designed for Sanger sequencing data cannot be used any more. Currently assembling DNA sequences of large genomes with complex repeating patterns like the human genome is not completely possible using NGS technologies. Assembly results obtained from NGS data are far less accurate than Sanger sequencing assemblies, even though the algorithms are more complex and better developed. Besides, by rapid improvements in NGS technologies, there has been much interest in sequencing DNA molecules of new species, however there is no perfect DNA Assembly algorithm to produce high quality results especially in the case of being *de novo* working on new species without having any knowledge about the resulting DNA sequence. Therefore, there has been much demand for new DNA assembly algorithms, fast techniques and methods to check the quality of DNA assemblers.

1.3 Summary

This chapter covered basic information about the DNA molecule, its structure and basic blocks as well as a brief introduction to DNA sequencing technologies and two types of currently available sequencing methods: Sanger Sequencing and Next-Generation-Sequencing. Each method's specifications and limitations are presented and the shotgun sequencing technique used to create datasets for DNA assembly problem is explained. The next chapter specifically presents the *DNA assembly problem* and introduces current approaches to solve it.

1.4 Organization of Thesis

The remainder of the thesis is organized as follows:

In chapter 2, different approaches to the Genome Assembly Problem including OLC, de Bruijn and Greedy methods are introduced.

Chapter 3 discusses the details about our new algorithm to solve the Genome Assembly Problem and introduces the new methods that we use compared to other assembly tools investigated in this thesis.

Chapter 4 includes the experimental results for running our new algorithm on several datasets and compares the performance of our algorithm to other DNA assemblers.

And finally, chapter 5 concludes the work that is done in the thesis and introduces the next steps and future work for this research.

Chapter 2

Review of *de novo* Genome Assembly Algorithms

Genome assembly is the process of finding the unique single and contiguous sequence of a DNA molecule by using its set of reads containing smaller sequences from random locations of the genome. For better understanding, DNA assembly can be compared to having many copies of a book which is only written with four characters (A, C, G, T), each of them passed through a shredder with different cutters, and aiming to obtain one clean copy of the book from the shredded parts [NSW+13]. Besides the obvious difficulties of the problem, more hidden issues should also be considered: the original book may contain repeated paragraphs, some shreds are modified through out the shredding process therefore having typos and shredded parts may be read from left-to-right or right-to-left (this is only specific to DNA reads

not the book example). Having a full DNA assembler capable of solving the problem for any input dataset is demanded by researchers, however based on our knowledge, such a system has not been created yet. This inability stems from several reasons:

- Different sequencing technologies have different characteristics [SJ08]. Some produce longer reads, making it easier for assemblers to detect overlaps, while some produce shorter reads with considerably high coverage, making the assemblers' work more difficult since they must deal with massive inputs with short lengths. Moreover, noise distributions are different among sequencing technologies [KSS+10]. Some technologies tend to produce noise at the starting and ending locations of reads, and some tend to generate noise in regions containing special sequences, such as long runs of homopolymers. Currently, creating a framework capable of addressing all of the mentioned situations and having significant performance for any sequencing technology seems impossible.

- Different species or even different individuals in the same species have different genomes. Genomes can be straight forward to assemble or can be extremely complex. Repeating patterns are the most important factor defining the complexity of genomes. If repeat lengths are less than reads size, there is a good chance of obtaining DNA fragments by resolving the repeat, however complex genomes have repeats of length far greater than actual read size, making them very difficult to solve

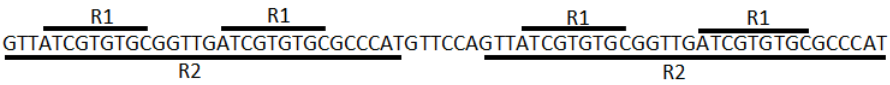

Figure 2.1: Two types of repeats in genome. Sequence *ATCGTGTGC* marked as *R1* is repeated four times through out the genome and it is resided in a bigger repeat pattern *GTTATCGTGTGCGGTTGATCGTGTGCGCCCAT* marked as *R2*

[MKS10]. Moreover, repeats can happen in the middle of one another. Figure 2.1 shows two different types of repeating patterns in a genome. Assemblers are usually tuned heuristically to target special types of genomes with some definite repeating patterns, making them incapable of solving the DNA assembly problem for any newly sequenced genome and also being *"de novo"*, not having any information about the genome.

- Some assembly methods that work for small sequencing projects are not scalable to large sequencing project dealing with very large genomes, having hundreds of millions of reads [LLS+11, LZR+10].

Conventional DNA Assembly algorithms were designed to work with Sanger-based sequencing reads. Sanger reads are more accurate compared to NGS reads and are long enough to ease the assembly process. Many assembly algorithms dealing with Sanger reads use the Overlap-Layout-Consensus (OLC) approach which will be explained thoroughly in section 2.1. However by invention of NGS technologies, sequencing new species becomes available while DNA assembly problem becomes more complicated. This is because NGS reads are not long enough to cover complex repeat structures in the

genome and are not very accurate compared to Sanger reads. New methods have been devised to specifically address assembly of NGS reads. Using de Bruijn graphs as a data structure is the most commonly used technique to tackle the DNA assembly problem and was first proposed by Pavel Pevzner in 2001 [PTW01]. This chapter covers three general techniques for solving DNA assembly problem. Section 2.1 describes the Overlap-Layout-Consensus (OLC) approach, section 2.2 explains the de Bruijn graph approach and section 2.3 presents greedy graph algorithms to solve the DNA assembly problem.

2.1 Overlap-Layout-Consensus (OLC) Methods

The Overlap-Layout-Consensus method is considered as the first approach proposed to solve the *de novo* DNA assembly problem. It was widely used in the Sanger reads era and it was proposed by having Sanger sequencing characteristics in mind. Celera Assembler [MSD+00], Arachne [BJS+02, JBG+03], CAP and PCAP [HY05] are among the most used OLC DNA assemblers. It is argued in [Pop09] that the OLC approach may not be scalable to be used for NGS data mainly because of being very time and memory intensive in the overlapping phase.

Three general steps should be performed in every OLC based assembler:

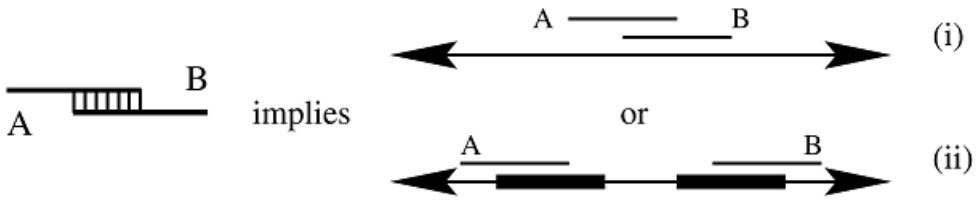

Figure 2.2: Two different scenarios are conceivable when two reads have overlap: (i) overlap is true, denoting a correct connection between the reads. (ii) overlap is denoting a repeating pattern and not expressing a direct connection between the reads. Detecting which condition the overlap denotes is usually not possible. (Image source: [MSD+00])

- **Overlap:** Find overlaps between all pairs of reads in input dataset. These overlaps make the main graph data structure to work on. Graph nodes represent reads and edges represent the overlap between reads. Overlapping criteria can vary in length and similarity percentage in different assemblers. Overlaps computation is the most time-intensive phase of the OLC approaches, requiring time proportional to the square of the number of reads, in the worst case (each read must be compared to all other reads, leading to $\binom{n}{2}$ operations)[Pop09]. However there are techniques to reduce the running time by parallelizing the computation and using multi-processor machines [Pop09]. Figure 2.2 shows two different scenarios in which two reads can have overlap and Figure 2.3 shows a simple overlap graph for a set of reads.

- **Layout:** The overlap graph usually becomes extremely large and complex. Thus a simplification phase should be done after the overlap phase

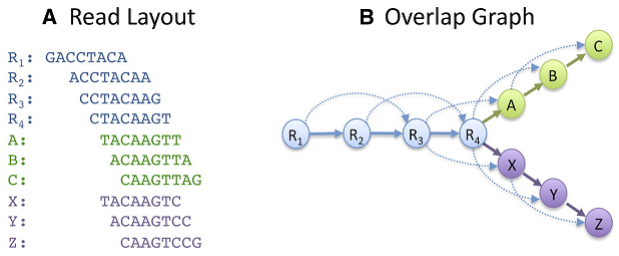

Figure 2.3: (A): set of reads with indentions showing overlaps between them. (B): overlap graph created for the read set which is usually used by OLC methods (Image source: http://genome.cshlp .org/content/20/9/1165)

to merge nodes that have unique overlaps, therefore the graph becomes smaller without losing any information. This phase is called *Layout*. By performing Layout algorithms, some graph nodes are merged together and unique sequences from the genome called *Contigs* are created. The output graph still can be seen as an overlap graph but between the contigs. Figure 2.4 shows a Layout scenario and formation of contigs.

- **Consensus:** The consensus phase aims to convert the whole graph to a single continuous sequence called a *Scaffold* representing the sequence of base-pairs that the input set expresses. This task can be done by finding a Hamiltonian path which traverses all nodes in the graph. A Hamiltonian path in an undirected graph is a path that visits every vertex (node) in the graph exactly once. Finding if a Hamiltonian path exists in a graph is *NP-Complete* [GJT76] which is a draw-back of using OLC methods for DNA assembly. Scaffolds can contain gap base-pairs and they connect contigs together by using mate-pair information.

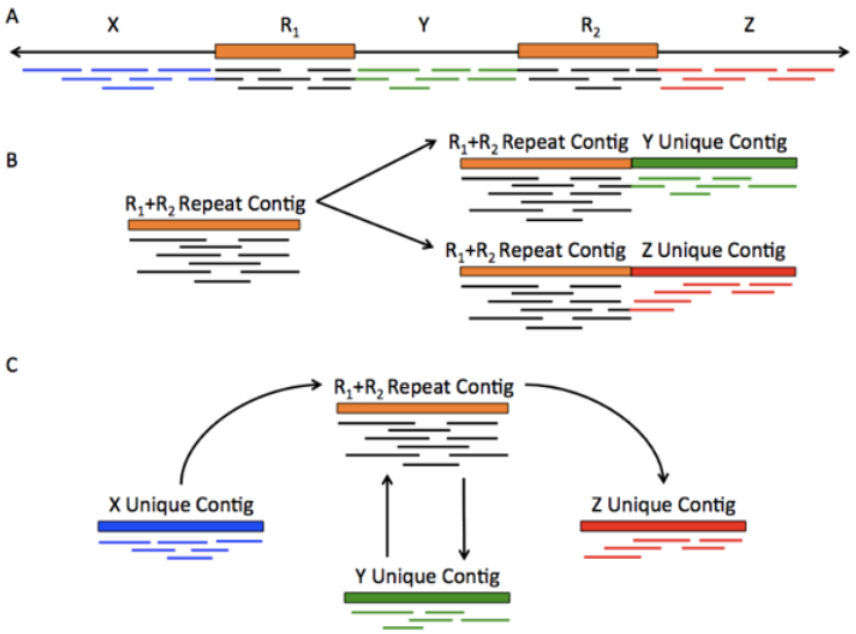

Figure 2.4: Layout scenario. Reads that have their connections determined are merged together and only nodes facing fork situations are left. Contigs are created by merging the nodes together. (Image source: http://gcat.davidson.edu/phast/olc.html)

Locations between the contigs are filled by gaps representing unknown bases, if they cannot be determined. Therefore, if there is not enough mate-pair information it is not possible to obtain one single scaffold.

The Overlap-Layout-Consensus technique is described in more depth in [MKS10, Bat05, PPDS04]

2.2 De Bruijn Graph (DBG) Methods

Generally the de Bruijn graph is a directed graph representing overlaps between sequences of symbols. The idea of using de Bruijn graphs to solve the DNA assembly problem was first proposed by Pavel Pevzner in 2001 [PTW01]. Currently de Bruijn graphs are the most commonly used technique to solve the DNA assembly problem for NGS data. There are various implementations and several DNA assemblers that are designed based on de Bruijn graph structure.

Pevzner [PTW01] defines the de Bruijn graph used for DNA assembly problem as follows: Given a set of reads $S = \{s_1, s_2, ..., s_n\}$, the de Bruijn graph $G(S_l)$ with vertex set S_{l-1} (the set of all $(l-1)$-tuples from S) is defined as follows. An $(l-1)$-tuple $v \in S_{l-1}$ is joined by a directed edge with an $(l-1)$-tuple $w \in S_{l-1}$, if S_l contains an l-tuple for which the first $l-1$ nucleotides coincides with v and the last $l-1$ nucleotides coincides with w. With this definition, if S contains only one sequence s_1, then the assembly is obtained by a path visiting each edge of the de Bruijn graph , a Chinese Postman Path [Fle90]. The Chinese Postman Path then can be translated to finding a path visiting every edge of a graph exactly once, an Eulerian Path Problem [Pev00]. This transformation happens by introducing multiplicities of edges in the de Bruijn graph. For example, every edge in the de Bruijn graph can be substituted by k parallel edges for every l-tuple repeating k times in s_1 [PTW01]. For real situations, the de Bruijn graph becomes very large and having errors in sequenced reads make the graph even more complicated. Even with error-free cases, the graph becomes very complicated. Thus the information about which l-tuples belong to the same reads is being used again to define Read-Paths and Eulerian SuperPaths introduced by [PTW01]. More information about the theories and detail specification of de Bruijn graphs used for DNA assembly problem can be found in [PTW01].

There are two significant advantages of de Bruijn graphs compared to the OLC technique that makes them practical for large genome projects:

- No need to precisely calculate overlaps between all reads.
- The idea proposed by Pavel Pevzner [PTW01] to use the Eulerian path to solve the DNA assembly problem instead of using the Hamiltonian path. An Eulerian path is a path that visits every edge in a graph exactly once. This makes a huge impact on DNA assembly problem as efficient algorithms in polynomial times exist to calculate Eulerian paths in graphs [AIS84, AV84, UTK88].

De Bruijn graph assemblies do not explicitly calculate every single overlap between all pairs of reads in the input dataset. They work based on *k-mer* calculation instead of read overlaps. All reads are first processed to find all overlapping substrings of length k. These substrings are called *k-mers*. All *k-mers* from all reads in the dataset are extracted and each *k-mer* is stored in memory only once, although it can be repeated in several reads. Fast data structures e.g. hash tables can be used to store and retrieve *k-mers*. A de Bruijn graph is created based on the *k-mers* set. Graph edges are the actual *k-mers* which are substrings of size k within the reads and graph nodes represent substrings of length *(k-1)* within the reads. Edges are established between any two nodes that have their *(k-2)* prefix and suffix in common. Figure 2.5 shows a de Bruijn graph for a sample consensus sequence with $k = 4$.

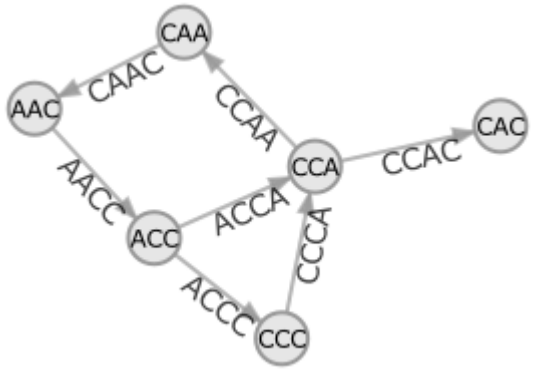

Figure 2.5: Simple de Bruijn graph with $k = 4$ for a set of reads that creates the consensus sequence "ACCCAACCAC" (Image source: http://gcat.davidson.edu/phast/debruijn.html)

The above definition creates the basic de Bruijn graph for DNA assembly, however different assemblers may have slightly different structures, definitions and assumptions to build the graph.

As reads are not considered as nodes in the de Bruijn graph and each unique *k-mer* is only stored once in the graph, de Bruijn graphs grow linearly with the input dataset size, making the DNA assembly problem solvable for large genomes. *K-mers* are usually stored in fast hash table structures in order to make the graph creation process as fast as possible. Moreover, *k-mers* are presented by graph edges and not nodes, therefore the final sequence can be extracted by finding an Eulerian path in the graph traversing all edges and not Hamiltonian paths traversing all nodes. This makes a huge impact on DNA assembly problem as efficient algorithms exist to calculate Eulerian paths in graphs [AIS84, AV84, UTK88].

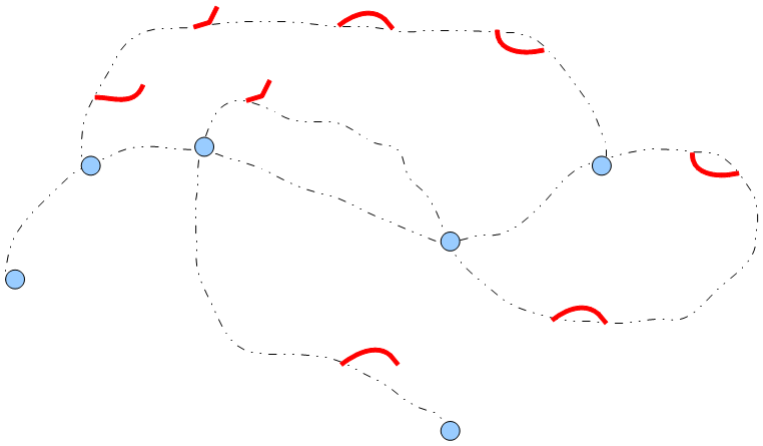

Figure 2.6: Tips and bulges in de Bruijn assembly graphs shown in red. Tips are branches in the graph that end without connecting to other parts of the graph. Bulges are branches from a node that come back to the main path after passing several edges. Bulges can be small, large or complex containing other bulges. (Image source: http://www.homolog.us/)

As for overlap graphs in OLC methods, de Bruijn graphs also become very large with millions of nodes for large genome assemblies. As de Bruijn graphs are based on *k-mers*, errors and noisy base-pairs in the dataset have significant influence on the graph as they produce different *k-mers*. In other words, de Bruijn graphs are more sensitive to sequencing errors than the overlap graphs. This makes the error detection procedure very important. One should also keep in mind that DBG methods are usually used with data generated from NGS technologies which are normally more error-prone. Assemblers usually define different types of errors and try to detect them after creating the graph. Errors are of different types including base insertion, base deletion and base replacement. Errors that occur at the end of the reads usually create tips in the de Bruijn graphs that are branches that end in a dead-end situation. Errors which occur in the middle of reads usually create bulges in the graph. These two types of graph structures are detected by assemblers and resolved before finding the Eulerian path in the graph. Differentiating between errors and repeat structures is usually not possible in most cases, therefore assemblers try to detect noisy parts by heuristics. Figure 2.6 shows tips and bulges in a de Bruijn graph.

After the graph simplification phase, an Eulerian path in the graph defines the result sequence for the assembly. However, there are fork situations in the graph which are nodes with out-degree of more than one which may create more than one Eulerian path in the graph. Not all Eulerian paths in the graph points to correct assembly. Assemblers use heuristics in order to find

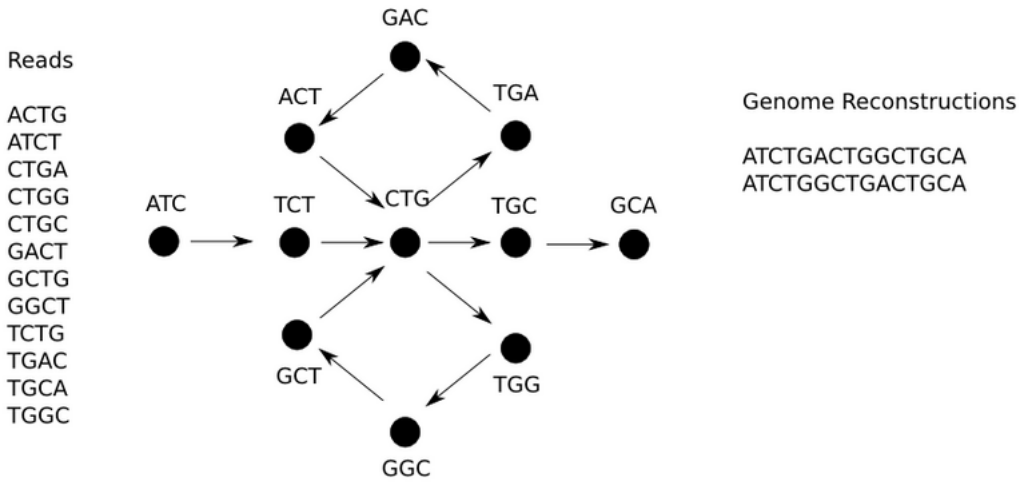

Figure 2.7: Two different Eulerian paths are conceivable for one set of reads. (Image source: http://sourceforge.net/apps/mediawiki/contrail-bio/index.php?title=Contrail)

the path which expresses the correct assembly. Heuristics used by different assemblers vary in this phase which makes every assembler somehow unique. Figure 2.7 shows two possible Eulerian paths in a de Bruijn graph, created for one unique set of reads. This is happening because some nodes in the graph can have more than one out-going edge that may not converge later (see node labeled CTG in figure 2.7). Having these type of nodes causes more than one Eulerian path to exist in the graph, however only one Eulerian path correctly represents the genome. For more information about the de Bruijn graph techniques refer to [PTW01, MKS10, Pop09].

2.3 Greedy Graph Methods

Greedy methods for the DNA assembly problem are based on one objective which is to choose the best overlap match at the current state of the algorithm. Reads with the highest overlap score are selected and merged together [Pop09]. The process continues until no more overlaps can be found. Large genomes sequenced using NGS technologies like the human genome are shown to be very complex to assemble, therefore it is not possible to solve them with greedy algorithms. Greedy algorithms usually get stuck in local maxima and are not be able to provide complete assemblies when dealing with sophisticated situations. However, they do not have any overhead in computation and time and are usually very fast. TIGR [SWAK95] and CAP3 [HM99] are among the first assemblers using greedy methods and SSAKE [WSJH07], SHARCGS [DLBH07] and VCAKE [JRB+07] are among the newer attempts to solve the DNA assembly problem with a greedy approach.

Recently there has been a renewed interest in using greedy methods in different parts of the DNA assembly problem and it is making significant progress. For example, Chikhi et al. [CL11] use a greedy based algorithm for their localized assembly algorithm to create scaffolds directly from reads. They unified the process of contig creation from reads and scaffold creation from contigs to one phase of creating scaffolds from reads. The Meraculous [CHS+11] assembler also uses a greedy approach to create contigs from input reads with newer techniques that leads to comparable results without having

huge computational overheads. This thesis also tries to improve assembly results by having a greedy view to the problem which will be explained thoroughly in chapter 3.

2.4 Summary

This chapter covered a literature review of the *de novo* DNA assembly problem. It first introduced the DNA assembly problem, its specifications and current limitations that assemblers deal with. Three different approaches to solve DNA assembly problem including Overlap-Layout-Consensus (OLC), de-Bruijn Graph (DBG) and Greedy methods are described. Specifications and limitations of each method are also presented for the two major types of DNA sequencing technologies. The next chapter presents our contig creation algorithm for NGS technology reads.

Chapter 3

New Contig Creation Algorithm

As already discussed in the previous chapter, the DNA assembly problem is generally solved with heuristics in mind. These include de Bruijn graph simplification or greedy-based techniques that decide on the correctness of graph edges heuristically. Different heuristics result in fragmented assemblies from different locations of the genome. By applying different heuristic and simplification methods, various assemblies can be generated for one genome and the problem becomes worse when it is infeasible to accurately select the best result. This is mainly because in *de novo* assembly there is no reference genome to match the results against. Results with higher length-based metric values such as the *N50* parameter are currently considered better assemblies, because they are producing larger fragments from the genome.

N50 value is a statistical measure of a set of numbers in which all elements of greater than or equal to *N50* value are covering at least half of the total addition of all set elements [MKS10]. *N50* is used in DNA assembly as a metric to measure quality of results. Larger *N50* values express on having larger contigs.

However there are experimental results [MPC+13, BFA+] that show larger contigs do not necessarily mean improved results and can be misleading when not correctly assembled. For instance, a new technique for evaluating genome assemblers [MPC+13] first splits the contigs/scaffolds on locations for which left and right pieces map onto distant locations in the base genome and then calculate the *N50* based on the split contigs, leading to more accurate calculations by skipping false positive links in assemblies. Such techniques essentially prevent the results from becoming biased by heuristics that accept many false positives during the assembly process. In this thesis, we also use a similar technique to first split the contigs from the locations that are not mapped to close locations in the test reference genome and then calculate the *N50* values.

3.1 Objectives

There are three main objectives in this thesis:

(1) Assemble fragments of the genome with the highest probability of correctness by avoiding the use of aggressive heuristics. Whenever there

is more than one way to extend contigs based on the *k-mers*, instead of selecting one direction and continuing the process, we terminate the contig creation procedure to be sure about contigs' quality and correctness. By having such a behaviour, we end up having smaller contigs in some datasets compared to other assemblers, but we can be certain that our contigs are perfectly matched with the target genome. We compensate for the small size in contigs by running the algorithm in parallel for multiple k values and combine the results from different runs at the end in order to obtain better lengths.

This objective is thoroughly explained in section 3.2.

(2) Provide the ability to run the algorithm with different k parameters. As described in chapter 2, reads are split into overlapping segments of length k to create *k-mers* which are the main inputs of the assembly algorithm. The k parameter has significant influence on assembly results and due to a variety of reasons including uneven data coverage, noisy data and varying repeat structures in different genome locations, a single value for parameter k does not necessarily give the optimal result for all locations in the genome. Having a very large value for k results in false positive links in fragments, while a small value for k results in tangled graphs which makes the problem impractical to solve [BNA+12]. Running the algorithm with different k values helps in generating considerably large and correct contigs from all locations of the genome. However, assembly

algorithms are very time- and space-consuming and it is not feasible to run multiple instances of the algorithm with dedicated memories in parallel. Trying to devise structures for multi *k-mer* assembly is a possible key to solving this problem.

This objective is thoroughly explained in section 3.3, and experimental results in chapter 4 shows the influence of using multiple values for k on the quality of results.

(3) By generating contigs with different k values from the genome locations that are usually left by other assemblers (because of using only one k value), there is a good chance of expanding contigs that are generated by other tools in order to obtain better results. Investigating the possibility of linking other tools' contigs to generate high quality contigs is the main target for this section.

This objective is thoroughly explained in section 3.3.1 and results from merging contig sets together are presented in section 4.3.

3.2 Producing Contigs

One of the most challenging problems in *de novo* DNA assembly is to find a good metric to measure the quality of created contigs. For new species that have not been sequenced before, there is not any reference sequence available to be used for verification purposes. In the absence of the reference genome,

there are length-based metrics such as the *N50* value which is widely used by assemblers to express the quality of results.

It is worth noting the critique in [MPC+13, BFA+] that larger contigs which lead to better *N50* values do not necessarily mean better results in terms of accuracy and also that there can be many false positive links in generated contigs.

We are also using the *N50* value in order to measure our quality of results. The detailed explanation on how to calculate more realistic *N50* values is presented in chapter 4.

This thesis focuses on using methods which are more conservative in expanding contigs and do not attempt to create larger contigs by lowering the certainty of contigs. The same idea is also proposed by [CHS+11]. Our approach is based on the method first proposed in [CHS+11] and it improves the results significantly by performing some changes to the algorithm flow and a new implementation which are all described in this chapter and appendices. Moreover, we use our generated contigs in order to improve results from other tools by importing their outputs to our system.

The assembly process can be described as follows:

(1) Stream the input data files to memory, store reads and pairing information. Fill in data structures for reads and *k-mers* and load the configuration files provided by the user. This part is explained in more detail in section 3.2.1.

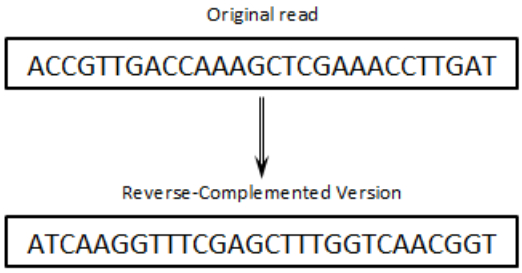

Figure 3.1: Reverse-complemented reads are generated by processing the original read backwards and changing any base character to its complementary base. (A $\leftrightarrow$ T , C $\leftrightarrow$ G)

(2) Extract *k-mers* while processing each read from the input files by having a pre-defined k value. Use hash-tables to store *k-mers* and their occurrence positions in the read set. Because it is not possible to determine which DNA strand the reads and corresponding *k-mers* belong to, all reads are processed to generate their reverse-complement as well. This doubles the input data space but boosts the quality of the results significantly. Contigs that are the reverse-complement of each other are filtered at the end of the assembly process by assuming that they are expressing the same location in the genome. Figure 3.1 depicts an example read and its reverse-complement.

(3) Detecting "noisy" *k-mers*, which are hash-table entries that occur less often than a fixed threshold number in the whole set of reads. The assumption behind this noise detection technique is that the input reads are randomly distributed through the genome with roughly even

Figure 3.2: For a read of length n, the right overlap (postfix) is a read for which its base-pairs positions 1 to $n-1$ are matched to the original read's base-pairs from positions 2 to n. Also, its left overlap (prefix) is a read for which its base-pairs from positions 2 to n are matched to the original read's base-pairs from positions 1 to $n-1$.

coverage, therefore all *k-mers* should be seen at least some minimum number of times, and entries that are seen less often than the threshold value can be assumed to be noise. This threshold value can be estimated to be lower but close to equal to the genome coverage depth of the input dataset, because it is assumed that each base-pair in the genome is roughly seen C times where C is the coverage depth. This part is explained in more detail in section 3.2.2.

(4) Find *(k-1)* length overlaps between all *k-mers* and link *k-mers* that can be the prefix or postfix of each other. An example of prefix and postfix *k-mers* (left and right links) are shown in figure 3.2. This part is explained in more detail in section 3.2.3.

(5) Extract *k-mers* that are expressing on unique base pair extensions either on their right or left links. These unique extensions become the

base information to create contigs based on k-mers. This part is also explained in more detail in section 3.2.3.

(6) Create contigs based on qualified *k-mers* with unique extensions until reaching dead-end or fork situations. This part is explained in more details in section 3.2.4.

(7) Analyse generated contigs from different k values (which can be run in parallel) to find any promising overlap between them. Because of having sequences from both DNA strands in the input set, contigs are made from both strands in this step. Therefore reverse-complement contigs should be detected and only one of them should be kept. This part is explained in more details in section 3.3.

(8) Import external contigs from other tools and analyse them, aiming to expand them even more by finding if they overlap with our generated contigs. This part is explained in more detail in section 3.4.

Figure 3.3 depicts a high level view of the proposed assembly algorithm.

One of the most important aspects of our algorithm is the extensive use of quality scores during the assembly process. Also, this algorithm does not rely on external error detection and correction tools. Many DNA error detection tools are using these quality scores to prune the data and detect noise before starting the assembly algorithm. However there are also some tools (e.g. [CHS+11]) that do not rely on external error detection tools. We

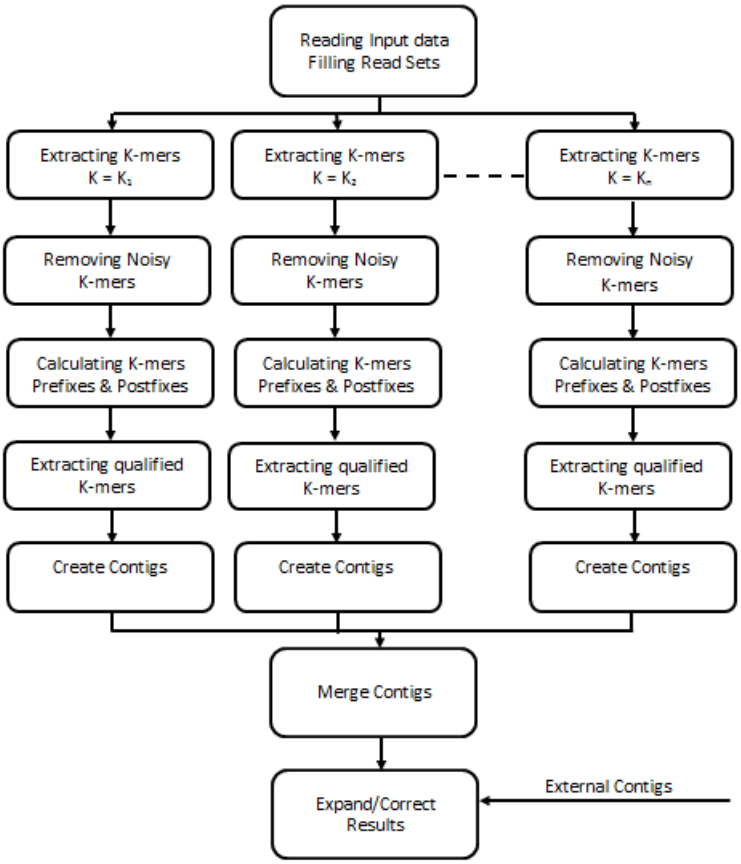

Figure 3.3: Assembly High Level Procedure

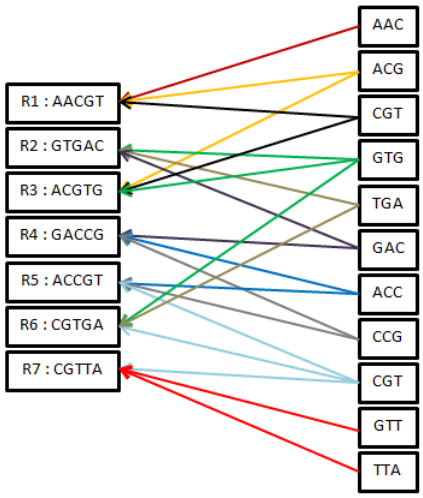

Figure 3.4: One unique *k-mer* may appear in more than one read. *k-mers* that are seen less than a pre-defined threshold amount can be treated as noise and filtered out.

use a method first described in [CHS+11] that handles the noisy parts of data based on the occurrence frequency of *k-mers* in input reads. We believe that in addition to saving reasonable time and space by avoiding the running of error detection tools, this approach also leads to better and more accurate results which experiments also support in section 4.2.1. The minimum acceptable frequency of *k-mers* in reads can be adjusted by the user. Figure 3.4 shows how *k-mers* may appear in more than one read.

Our algorithm obtained its basic idea from the research in [CHS+11] and works to improve the quality of results. The differences between our algorithm (and implementation) and [CHS+11] can be summarized as follows:

- The Meraculous assembler [CHS+11] only considers *k-mers* which have

unique extensions in the contig creation process. Although this is correct and generates very high quality contigs, it can be improved by adding contigs that are expressing on unique extensions with the probability of more than a threshold value; therefore we can use *majority vote* on the unique base-pair extensions and the number of trusted *k-mers* increases. This consequently leads to larger contigs while keeping the quality of contigs very high. There are also situations in which one end of a *k-mer* expresses a "harsh fork" situation in which it cannot be resolved even by majority voting but the other end is resolved. This will be discussed further in section 3.2.3. These *k-mers* are also not being used by the Meraculous package but can be added to the trusted *k-mers* list in our implementation because they help to create larger contigs with comparatively high quality to other tools.

- Different data structures and hash functions are used in our tool to produce better results in comparison to the Meraculous assembler's implementation. Section 4.2.2 shows our tool's improvements in comparison to the Meraculous package.

- The Meraculous assembler is not capable of running the algorithm for different k values in parallel, thus it has difficulties creating enough large contigs from all genome locations on the datasets in our experiments. Our tool is capable of working with different k values in parallel and does create comparably large contigs from all locations of the genome.

Experimental results support this idea and show the improvement when using multiple k values.

- Our tool is also designed to accept other assemblers' contigs in order to analyse and expand them. There is no feature similar to this in the Meraculous package.

3.2.1 Input Data Loading and *Reads*/*K-Mer* Class Structures

All assemblers should be able to deal with large input files. It is assumed in this thesis that inputs are coming with pair information showing which two reads are connected as pairs. Algorithms are designed for Illumina technology reads and input sequence data must be in .fastq file format, however other file formats can also be easily supported by adding appropriate parser code for them. In the case of .fastq file format, there are an even number of files each including read information for one set of pairs. Two files that are presenting pair information must have an equal number of reads. Figure 3.5 shows a sample configuration file that includes addresses for .fastq files and Appendix A shows a sample set of input files in .fastq format.

Read objects are created by processing the input data. Quality scores are also stored and pairing information is set for all reads. The main algorithm does not work directly with these sequences and they are only used once to create *k-mer* sets, therefore reads can be removed from the memory after

```
<?xml version="1.0" encoding="utf-8" ?>
<AssemblyParameters>
  <AssemblyParameter key="ReadSet1Address" value="C:\real_datasets\100k_chr10_60-160k
                    \100k_chr10_60-160k.1.fq"></AssemblyParameter>
  <AssemblyParameter key="ReadSet2Address" value="C:\real_datasets\100k_chr10_60-160k
                    \100k_chr10_60-160k.2.fq"></AssemblyParameter>
  <AssemblyParameter key="RefrenceGenomeAddress" value="C:\real_datasets\
                    100k_chr10_60-160k\
                    100k_chr10_60-160k.ref.fa"></AssemblyParameter>
  <AssemblyParameter key="ContigsOutputAddress" value="C:\my_results\
                    100k_chr10_60-160k\
                    100k_chr10_60-160k_contigs.txt"></AssemblyParameter>

  <AssemblyParameter key="ConsiderPairReads" value="true"></AssemblyParameter>
  <AssemblyParameter key="MultipleKs" value="true"></AssemblyParameter>

  <AssemblyParameter key="Ks" value="41,31,19"></AssemblyParameter>
  <AssemblyParameter key="ContigsMinLength" value="82"></AssemblyParameter>
  <AssemblyParameter key="MultiplicityMinThreshold" value="10"></AssemblyParameter>
  <AssemblyParameter key="HighQualityMinThreshold" value="20"></AssemblyParameter>
  <AssemblyParameter key="ContigsOverlapValue" value="10"></AssemblyParameter>
  <AssemblyParameter key="MajorityVotingThreshold" value="0.6"></AssemblyParameter>

  <AssemblyParameter key="ResolveNotUUExtensions" value="false"></AssemblyParameter>

  <AssemblyParameter key="BaseQualityValue" value="64"></AssemblyParameter>

</AssemblyParameters>
```

Figure 3.5: A sample configuration file for DNA assembly algorithm.

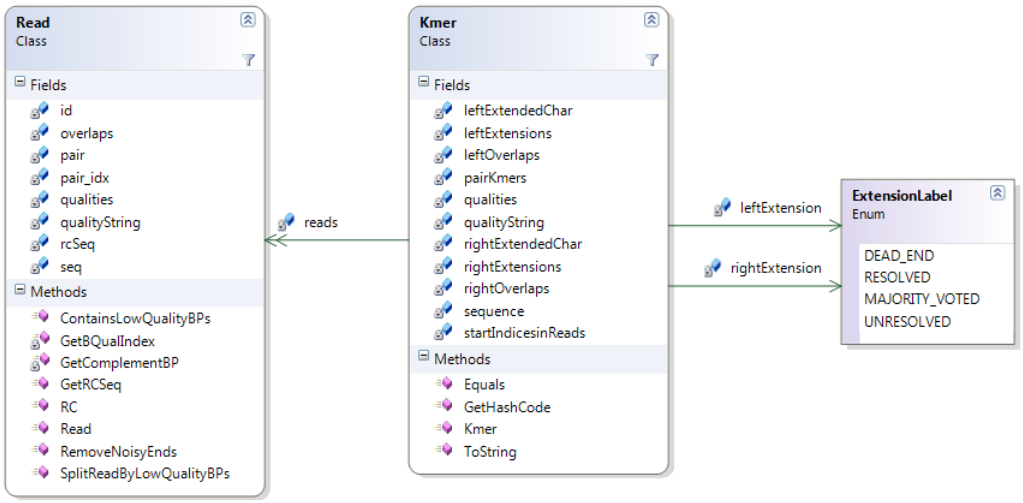

Figure 3.6: Class diagram showing *Read* and *K-mer* class structures. Each *K-mer* has list of *Read* objects in which it is belonged to. *K-mer* ending labels are also presented with an Enumeration class.

the *k-mer* creation process. Figure 3.6 shows the class diagram for the *Read* and *K-mer* classes. Appendix B provides complete information regarding the class hierarchies, structures and implementation details.

The main algorithm can work with multiple k values. Each k value has its own *k-mer* set which is created based on the input reads. .NET framework hash-table structures are used to store *k-mer* sets. The hash function used in our tool is the algorithm presented by Jon Skeet [Ske13] for generating hash codes for byte arrays presented in algorithm 1. By one-time processing of reads all *k-mers* and their occurrence counts are extracted and stored in the hash-table. Each *k-mer* also keeps track of the reads that contain it. This information is used in the next steps to remove noisy *k-mers*.

```
hash = 17;
//Cycle through each element in the array.
foreach (byte b in bytes)
{
    //Update the hash.
    hash = hash * 23 + b.GetHashCode();
}
return hash;
```

Algorithm 1: Jon Skeet's hashing algorithm used in this thesis.

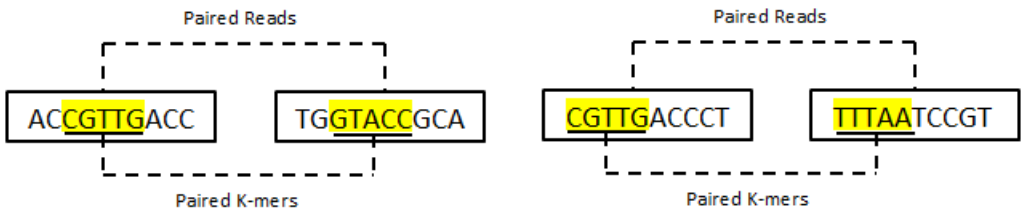

Figure 3.7: Paired *k-mers* are two *k-mers* in two paired reads. *k-mers* pairing relation is not unique. *k-mers* CGTTG is assumed to be paired with *k-mers* GTACC considering the left read pair but *k-mers* CGTTG can be seen in another read like the right read pair and it is assumed to be paired with *k-mers* TTTAA as well. ($k = 5$ in this example)

In the same way that reads keep the pairing information, *k-mers* also keep pair information with other *k-mers*, but with a slight difference. All pairing information between reads are unique but one single *k-mer* may have more than one pair *k-mer* because *k-mers* occur in more than one location in different reads. Basically it is assumed that if read A has a pair read of B then the first *k-mer* of read A pairs with the first *k-mer* of read B, and so on. As it is also depicted in Figure 3.7, this relationship is generally not unique. This type of information is also stored in data structures (described in Appendix B) and it may be useful for further analyses to use mate-pair information for the scaffolding problem which is briefly presented in 2.1.

In order to reduce memory usage, we store reads and *k-mers* information as compactly as possible. This is achieved by reserving only 2 bits for each base-pair in sequences as there are only four possible base-pair characters. "A" base-pairs are stored as 00, "C" base-pairs are stored as 10, "G" base-pairs are stored as 01 and "T" base-pairs are stored as 11. Therefore, each byte which normally should keep only one base-pair, actually stores four base-pairs in our program, resulting in reduction of memory usage by 75%. A library including encoding, decoding and other useful functions for compressing the DNA sequences is implemented in our tool which is presented in Appendix B.

3.2.2 Removing Less Frequent *k-mers*

Many assemblers use error detection/correction techniques to find and resolve noise in input data, and then run the assembly algorithm on the corrected

data. It is shown (e.g. in [KSS+10]) that using error detection/correction techniques improves the assembly results, however there are some problems using these methods:

- Error detection/correction tools may filter our correct data because of having lower coverage or any other complexity in the data. However this is inevitable and currently there is no other approach in our knowledge to address this problem.
- Error detection/correction tools are time demanding and their run time increases drastically when working with large inputs e.g. human genome, even though they just need to be run once.

Therefore in this research, we follow the idea from [CHS+11] to not use any error detection/correction tool beforehand and instead handle the noisy data in the middle of the assembly algorithm when creating contigs. In addition to having faster running time, it is also shown that this approach can lead to better and more accurate results [CHS+11].

By creating the *k-mer* set, the occurrence number of every single *k-mer* in the read set is counted and stored in the hash-table structure. A minimum threshold can also be set by the user that defines the minimum occurrence number of *k-mers* in the input set. All entries that have fewer occurrences will be removed.

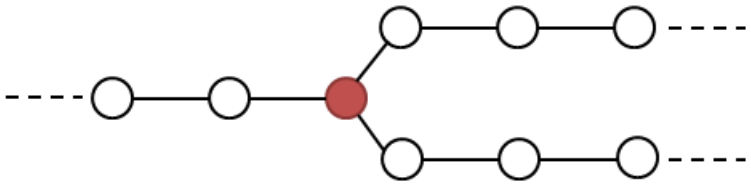

Figure 3.8: Each node represents a *k-mer* and each edge defines overlap between two *k-mers*. Nodes that have only one edge going in/out of them are considered qualified and will be detected by our algorithm. Some nodes such as the one labelled in red are in fork situations, meaning the algorithm cannot decide which *k-mer* succeeds it without using heuristics. Heuristics used to resolve these fork situations have drastic influence on assemblers' performance. These fork situations are the ones that could not be resolved by *majority voting* or other techniques.

3.2.3 Finding *k-mer* Overlaps

As described in the previous chapter, de Bruijn graph-based assembly methods create overlap graphs with each node containing a *k-mer* and each edge defining overlap between two *k-mer* nodes, thus defining sequences of length *(k+1)*. Using the whole *k-mer* set, this creates a very large and memory intensive de Bruijn graph which has many nodes and edges that prevent the algorithms from effectively simplifying the graph if it happens by using an inappropriate k value. The effect of using inappropriate k values in large de Bruijn graphs is explained in the previous chapter. In this research we follow the idea of not using the whole *k-mer* set to create the de Bruijn graph (as in [CHS+11, CL11] and only consider *k-mers* which are not involved in fork locations. Figure 3.8 shows qualifying *k-mers* and a *k-mer* in a fork situation which is not considered for the first round.

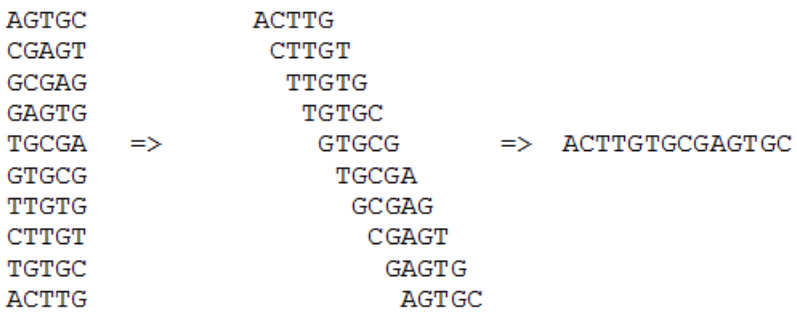

Figure 3.9: Overlapping *k-mers* connect together and create larger fragments from DNA. This simple example only shows how *k-mers* can have right and left overlaps and does not show repeat structures in the genome, therefore in this example one final unique sequence can be achieved.

In order to find qualifying *k-mers*, first all overlaps should be detected. The basic idea of detecting the overlaps is to find which two *k-mers* have similar prefix and suffix sub-strings of length *(k-1)*. If *k-mer A* has a prefix (sub-string from the first element to the one before the last element) equal to *k-mer B*'s suffix (sub-string from the second element to the last element), then *k-mer B* can connect to *k-mer A* on the right in order to make a *(k+1)-mer*. This extension can be checked from both ends to create left and right overlaps for all *k-mers* in the set. Only overlaps that have a quality score of more than a defined threshold in the overlapping base-pairs are considered in this step, which ensures skipping noisy data. Figure 3.9 shows how *k-mers* connect together.

By having all overlaps for every single *k-mer*, qualified *k-mers* must be detected. For this reason, *k-mer* ends are first labelled as follows:

- **(Resolved):** All left/right overlapped *k-mers* express on a unique extension base pair. Figure 3.10 shows a Resolved state scenario.

Figure 3.10: Resolve State. All high quality extensions express on base-pair *A*, selecting it as a true extension for the *k-mer*.

- **(Dead-End)** There is not any left/right overlap.
- **(Majority Voted)** Overlapped *k-mers* do not all express on a unique extension base-pair but the majority of entries vote for a unique extension with probability of higher than a defined threshold. Figure 3.11 shows a Majority-Voted state scenario.
- **(Unresolved)** If none of the above labels apply, the *k-mer*'s end is labelled as "unresolved" which shows a fork situation. Figure 3.12 shows an Unresolved state scenario.

k-mers that are considered to express on unique extensions in both their right and left overlaps ("Resolved" or "Majority-Voted" labels) are considered "qualified".

The idea of having labels including "Resolved", "Dead-End" and "Unresolved" for *k-mer* ends were first proposed by [CHS+11] but the "Majoriy-

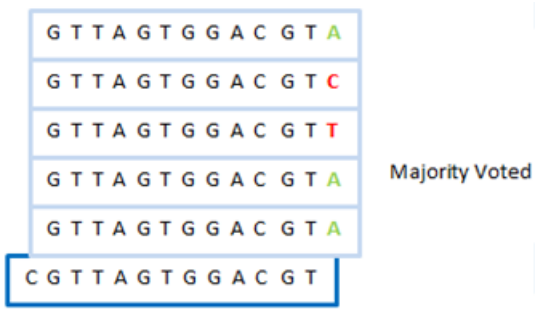

Figure 3.11: Majority-Voted State. Not all high quality extensions express on a unique base-pair but most of them express on base-pair A selecting it as a unique base-pair extension. Minimum probability for Majority-Vote can be set by the user.

Figure 3.12: Unresolved State. Not all high quality extensions express on a unique base-pair and none can be selected as a majority.

Voted" label is a contribution of this research to the community.

Qualified *k-mers* can build unique and uncrossed paths through the large de Bruijn graph that do not have any forking nodes, therefore the algorithm does not need much time and memory space in comparison to other tools that create the full de Bruijn graph at the first step. In this way the most important information is obtained from the de Bruijn graph without any need to build the whole memory-intensive graph which is not possible for large

datasets like the human genome.

3.2.4 Contig Creation

Qualified *k-mers* are the base information used in the contig creation process. Each qualified *k-mer* is expressing on a unique single-base extension in both its right and left links. Thus, two overlapping *k-mers* can be created by having one starting *k-mer*. Newly created *k-mers* are checked in the qualified *k-mer* set and if they exist, the base *k-mer* is extended by one base-pair (*k-mers* merged) and the process continues by following the extensions for the new added *k-mer*. The contig creation process terminates when both ends of the contig reach a dead-end or unresolved situation with nothing to match from the qualified *k-mer* set. Selecting the base *k-mer* to start is not important and can be done randomly. New contigs are generated until the qualified *k-mer* set runs out of elements. Algorithm 2 shows the procedure of creating contigs from qualified *k-mers*.

3.3 Multi *k-mer* Assembly Solution

Many current assembly algorithms consider a fixed value for k and this parameter has a significant role in obtaining the best results. There are methods to analyse the input data and find the most appropriate k value for the given input[SWJ+09, BMK+08], however, to the best of our knowledge, many of the proposed methods assume an even coverage through the input

```
while qualifiedKmers is not empty do
  cntg ← instantiate a new contig object
  firstKmer ← pick and remove first element from qualifiedKmers
  rightExtension ← firstKmer's rightExtension
  leftExtnesion ← firstKmers's leftExtension
  cntg ← firstKmer
  rightTruncated ← false
  leftTruncated ← false
  finish ← false
  while finish is not true do
    finish ← true
    rightOverlapKmer ← cntg[n − k : n] + rightExtension
    leftOverlapKmer ← leftExtension + cntg[0 : k]
    if rightTruncated ≠ true then
      if qualifiedKmers contains rightOverlapKmer then
        cntg ← cntg + rightExtension
        rightExtension ← rightOverlapKmer's rightExtension
        remove rightOverlapKmer from qualifiedKmers
        finish ← false
      else
        rightTruncated ← true
      end if
    end if
    if leftTruncated ≠ true then
      if qualifiedKmers contains leftOverlapKmer then
        cntg ← leftExtension + cntg
        leftExtension ← leftOverlapKmer's leftExtension
        remove leftOverlapKmer from qualifiedKmers
        finish ← false
      else
        leftTruncated ← true
      end if
    end if
  end while
  add cntg to contigs
end while
return contigs
```

Algorithm 2: Contig creation algorithm.

data and calculate a single k value for the data set; this is not always correct especially for human genome data because of its size and complexity in repeat patterns. Moreover, repeating patterns in the genome have different characteristics and they play the most important role in the quality of assembly results. Different k values result in either resolving repeat structures, or being stuck in the middle of the contig creation process, and there is not any unique k value that can work for all locations of the genome. Small k values make the de Bruijn graph very tangled and messy, thus the paths are not fully detectable and the quality of results decreases. On the other hand, large k values may resolve repeat patterns with length of less than k but may fail to detect overlaps between reads, particularly in low coverage regions, making the graph more fragmented [BNA+12].

There have been attempts in assemblers like [ZB08] to find the most appropriate k value and run the algorithms for multiple ks but the assemblers themselves do not try to improve the overall results based on outputs from multiple k values.

In this research, the most important goal is to produce qualified contigs from all over the genome using different k values. The idea of using multiple k values in order to build contigs is also proposed by other assembly tools (e.g. in [BNA+12, MPC+11]) but we claim to have a very simple way of doing this without any complicated mathematics and complex structures that brings overhead.

By having results for different k values, it is more likely that the best

contigs from all locations of the genome are being created even though they are from different runs. Therefore it is feasible to obtain larger contigs by analysing the results from different runs and trying to merge the overlapping parts. However, a significant portion of contigs from different k values are expressing on the same locations in the genome, therefore repeating parts should be detected and removed at the end.

3.3.1 Contigs Merging

Contigs are contiguous portions of the genome that the assembler successfully constructs. Because there is not any information regarding which strand the base reads belong to, contigs are created on both strands which brings two versions of each contig (the contig itself and its reverse-complement) to the contig set. However, contigs do not have any overlap of length more than k with each other, because if they had it would be detected in previous steps of the assembly algorithm, unless they come from different k runs. Therefore attempting to merge contigs all generated from one fixed k value does not improve the results, but the idea of merging works when dealing with contigs generated from different k values.

Different k values generate different contigs with different lengths through the genome. In some assemblies, more repeats may be resolved and different locations of the genome may be constructed. The main reason behind this is already discussed in section 3.3. Some locations of the genomes which do not have very complex repeat structures tend to be constructed with almost

List<Contig> oldContigs;
List<Contig> newContigs
for all cntg in oldContigs **do**
  $newCntg \leftarrow$ instantiate new contig object
  for $i = n - 1$ downto 0 **do**
    $newCntg[n - i - 1] \leftarrow complementBP(cntg[i])$
  end for
  add *cntg* to *newContigs*
  add *newCntg* to *newContigs*
end for
return *newContigs*

Algorithm 3: Creating Reverse-Complement Contigs.

every reasonable k value. Thus, contigs from different assembly runs do have overlaps and applying a merging technique should improve the results.

The first step to merge contigs is to find overlaps between all of the input contigs. As contigs can belong to each of the genome strands, reverse-complements are generated for all of them at the first step. By actually doubling the dataset, we can be sure to find overlap between two contigs that construct the same location in the genome but from different strands. Algorithm 3 shows how the contig set doubles in size when creating Reverse-Complement versions. In order to find extensions for the contigs, an algorithm is needed to check if there is any overlap between two input contigs or not. There are three situations in which two contigs can be linked together:

- (1): The first contig's ending base-pairs are matched with the second contig's starting base-pairs, thus the first contig can be linked to the second contig from the left. The Algorithm to check this condition is

```
p ← L1 − 1
while p ≥ CONTIGS_MIN_OVERLAP do
  match ← true
  for i = 0 to p − 1 do
    if cntg1[L1 − p + i] ≠ cntg2[i] then
      match ← false
      break
    end if
  end for
  if match then
    return cntg1 + cntg2.substr(p)
  end if
  p ← p − 1
end while
return null
```

Algorithm 4: Contigs left link check algorithm

```
p ← 0
while p + L1 ≤ L2 do
  match ← false
  for i = 0 to L1 − 1 do
    if cntg1[i] ≠ cntg2[i + p] then
      match ← true
      break
    end if
  end for
  if match then
    return cntg2
  end if
  p ← p + 1
end while
return null
```

Algorithm 5: Contigs substring check algorithm

$p \leftarrow L1 - 1$
while $p \geq CONTIGS\_MIN\_OVERLAP$ **do**
  $match \leftarrow true$
  for $i = 0$ to $p - 1$ **do**
    if $cntg1[i] \neq cntg2[L1 - p + i]$ **then**
      $match \leftarrow false$
      $break$
    end if
  end for
  if $match$ **then**
    return $cntg2 + cntg1.substr(p)$
  end if
  $p \leftarrow p - 1$
end while
return *null*

Algorithm 6: Contigs right link check algorithm

Contig cntg1;//cntg 1 is always the smaller contig
Contig cntg2;
$L1 \leftarrow length(cntg1)$
$L2 \leftarrow length(cntg2)$
$consensus \leftarrow RightLinkCheck(cntg1, cntg2)$
if $consensus \neq null$ **then**
  return $consensus$
end if
$consensus \leftarrow LeftLinkCheck(cntg1, cntg2)$
if $consensus \neq null$ **then**
  return $consensus$
end if
$consensus \leftarrow SubStringCheck(cntg1, cntg2)$
if $consensus \neq null$ **then**
  return $consensus$
end if
return *null*

Algorithm 7: Finding contigs overlap

presented as Algorithm 4.

- (2): The first contig is completely repeated in the second contig, thus the second contig expresses the merging result. The Algorithm to check this condition is presented as Algorithm 5.

- (3): The first contig's starting base-pairs are matched with the second contig's ending base-pairs, thus the first contig can be linked to the second contig from right. The Algorithm to check this condition is presented as Algorithm 6.

Algorithm 7 shows the procedure of finding the overlap between two input contigs (consensus sequence). It calls other procedures presented in Algorithm 4, Algorithm 6 and Algorithm 5 to check for all conditions in which two contigs can generate a consensus sequence. The maximum overlap length between contigs can be set in the assembler's configuration file and is usually equal to the minimum k value considered. By being able to merge any two input contigs, an iterative procedure can be devised to merge and extend contigs until no more extension is possible. Algorithm 8 shows this procedure.

3.4 External Contigs Expansion

Contigs created using the approach described in this thesis are assumed to express certain fragments in the genome with high probability. Running the assembly algorithm for different k values and merging the results from

```
while contigs > 1 do
  baseContig ← contigs[0]
  remove baseContig from contigs
  overlapFound ← false
  List < Contig > newlyAddedContigs
  for all cntg in contigs do
    consensus ← ContigsOverlaped(baseContig, cntg)
    if consensus ≠ null then
      remove cntg from contigs
      add consensus to newlyAddedContigs
      overlapFound ← true
      if consensus == cntg then
        break
      end if
    end if
  end for
  add newlyAddedContigs to contigs
  if overlapFound == false then
    addbaseContigtofinalContigs
  end if
end while
return finalContigs
```

Algorithm 8: Contigs expansion algorithm.

different runs usually leads to better results. While merging results from different runs of our own assembly algorithm is useful, importing contigs from other tools can also be very beneficial. The same set of expansion and merging algorithms can be performed on imported contigs too. However, this also creates false positive links between the contigs due to sequences in repeating regions. Currently, we detect the false links after the contig creation process by aligning and comparing the fragments to the human reference genome, and only consider the correctly aligned fragments for evaluating the algorithm. Devising techniques to prevent false positives during the merging algorithm is part of our future work for this research.

There are definitely some areas in the genome that are covered by other assemblers. Also different assemblers can construct different locations of one genome because of using different heuristics and assumptions. Therefore merging results from different assemblies should lead to better contigs. By having all contigs which are built from different k values, there is a better chance of creating larger contigs from state of the art algorithms while not reducing the contigs' correctness. The procedure of merging external contigs with our generated result is the same as the algorithm described in section 3.2.4. The experimental results in section 4.2.2 show that importing other tools' contigs to our system and performing the expansion algorithm can help obtain significantly better results.

3.5 Summary

This chapter described the main algorithms used in this thesis in order to create contigs from input short reads. Methods to load input data to the memory, storing them in the designed data structures and performing algorithms to create contigs are presented in this chapter. Moreover, running the assembly algorithm for multiple k values in parallel is described in section 3.3. Finally, we proposed a method to merge contigs from different assembly runs and the ability to utilize external contigs from other tools in order to improve their quality of results.

Chapter 4

Experimental Results

4.1 Experimental Results Terminology

To the best of our knowledge, the *de novo* DNA assembly problem for the human genome is still an open problem. It is discussed in [MPC+13] that when dealing with complex genomes, using different available assemblers may not help to obtain better results unless there is better input data with less noise, better coverage, longer reads, and etc. Therefore, it is believed that currently the most important problem is the data and not the algorithms. However algorithms also vary significantly: some are not even scalable to human genomes and others that are capable, obtain limited results compared to results from Sanger data.

The most widely used method to distinguish different assemblers is to measure their performance using length-based metrics such as *N50* described

in chapter 2. Larger *N50* value shows that larger contigs are created, which can primarily be considered as a better result. However, sometimes *N50* values can become misleading, when the generated contigs are not accurate. Unfortunately deciding if a contig is correct or not is currently impossible in *de novo* DNA assembly as there is no reference genome available to compare to.

For comparisons, we select our datasets in this thesis from the human genome, therefore we can use the human reference genome (hg19) in order to estimate the accuracy of the contigs and detect the false links between the final contigs. In order to detect the false links in the contigs, we split each contig from all locations that the left and right fragments are aligned to distant locations in the reference genome, meaning the contig is not built in a correct way and should split. In other words, we consider alignment blocks from the BLAT tool as the correctly mapped fragments and the maximum allowable gap between the alignment blocks is set to 50 bases. From now on, whenever we refer to the *N50* value, we mean the calculated value based on the fragments generated by splitting contigs in described locations, and not the base contigs which are the outputs of the assemblers.

There are two main sections in this chapter for our experimental results:

(1) **N50 comparisons:** These measure the quality of results based on contigs' length. Calculating the *N50* parameter is done by the formula given in chapter 3 page 27 and can be accomplished by only having the contigs' size and the targeting genome's size. Before calculating *N50*

values, contigs are split into several fragments as described above.

(2) **External contigs expansion results:** These show the quality of results when external contigs are added to our generated contigs and the expansion algorithm is performed on the dataset.

4.1.1 Datasets

Different locations of the human genome with different sizes are selected in 9 different datasets in order to perform experiments. The datasets used in this thesis are described in table 4.1.

Table 4.1: Experimental data sets.

Dataset	Genome Length	Location	Chromosome	Reads Count
1	1Kb	100K-101K	1	190
2	10Kb	100K-110K	1	3452
3	10Kb	60K-70K	10	1296
4	100Kb	100K-200K	1	19246
5	100Kb	60K-160K	10	17178
6	1Mb	100K-1100K	1	190030
7	1Mb	60K-1060K	10	182370
8	10Mb	100K-10100K	1	1766556
9	10Mb	60K-10060K	10	1825054

In order to find contigs' accuracy, all contigs are aligned to the hg19 reference genome using the BLAT tool [Ken02]. Among all possible alignments for each contig, the alignment which builds more unique fragments in the genome in the specific locations is selected. Alignments which are not in the

selected region or do not express on any new fragments that are not already filled by other contigs are filtered out.

4.2 Results

Three assemblers are selected to run on proposed datasets. Assemblers are:

- Meraculous [CHS+11]
- SOAPdenovo [Li09]
- Velvet [ZB08]

These assemblers are selected because of their popular use among researchers and their stability. The Meraculous [CHS+11] tool is specifically selected because of having a very close algorithm to our technique presented in this thesis.

Our tool is capable of running the assembly process for multiple k values in parallel with any k value set provided. Other tools either do not have this feature or have it implemented in a way that cannot accept all k combinations in one run, therefore we ran each assembler for each k value individually and get the average between the runs. The k values that are used in our experiments are fixed for all datasets, and cover a range of small and large values. These values are: k: 19, 31 and 41. It should be noted that small changes in the value of k do not have very much effect on the results obtained. These are typical values for k as used in other research when the read length

is 100 as in our case. We chose a range of values in our experiments to show how our algorithm works for different values. Note that if the read length changes, the k values should also be adjusted.

4.2.1 N50 Results

This section presents the *N50* values obtained for each assembler's run on the datasets. The values for other assemblers are the averages obtained from three different runs for selected k values. Tables 4.2 and 4.3 show comparisons between the *N50* results of assemblers for all datasets.

The Meraculous assembler is the closest assembler to our method in terms of the algorithms and heuristics. One of our main objectives was to outperform the tool that has the closest algorithm to our method. Results show that our tool has better performance than Meraculous in all of the experimented datasets. The reason why our tool out-performs Meraculous in all test-cases, and why it has the best performance in some of the datasets, can be explained as a result of using multiple k values in the assembly process. Different k values are producing reasonably large contigs from different locations in the genome and merging the results from various k runs, helps to obtain significantly better results in some cases. However this directly depends on the datasets' characteristics and repeat patterns which are not known before hand. In two of the datasets the Meraculous assembler has the *N50* value of zero which means the total length of all fragments is not more than half of the targeted genome, while our tool obtains *N50* values of 86 and 573. By

Table 4.2: N50 Results for Datasets #1 to #5. Best result for each dataset is bold

Assembler	**N50**	**Largest Fragment**
Data Set 1: 1K_Chr1_100-101K		
Our tool	772	772
Meraculous	208	288.6
SOAPdenovo	**889.6**	**889.6**
Velvet	540.6	540.6
Data Set 2: 10K_Chr1_100-110K		
Our tool	**1434**	**2679**
Meraculous	311.6	1275.6
SOAPdenovo	677.3	1655
Velvet	770.6	1733.3
Data Set 3: 10K_Chr10_60-70K		
Our tool	86	355
Meraculous	0	248.6
SOAPdenovo	**1138.6**	**1762**
Velvet	761.3	1719
Data Set 4: 100K_Chr1_100-200K		
Our tool	285	4438
Meraculous	54.6	1782.3
SOAPdenovo	**676.3**	**3752**
Velvet	509	3434.33
Data Set 5: 100K_Chr10_60-160K		
Our tool	269	2117
Meraculous	59.6	713.3
SOAPdenovo	**1447.3**	**3411**
Velvet	1074.6	4053.3

Table 4.3: N50 Results for Datasets #6 to #9. Best result for each dataset is bold

Assembler	**N50**	**Largest Fragment**
Data Set 6: 1M_Chr1_100-1100K		
Our tool	**573**	**2584**
Meraculous	0	3333
SOAPdenovo	117	6444.6
Velvet	109.6	5215.6
Data Set 7: 1M_Chr10_60-1060K		
Our tool	341	3077
Meraculous	64.3	1661
SOAPdenovo	**1429.6**	**9121.3**
Velvet	1135.3	6648.6
Data Set 8: 10M_Chr1_100-10100K		
Our tool	300	3255
Meraculous	21.3	3333
SOAPdenovo	**1002.6**	**9885**
Velvet	705	8882.6
Data Set 9: 10M_Chr10_60-10060K		
Our tool	387	3869
Meraculous	68.3	1810.6
SOAPdenovo	**1504.6**	**14139.3**
Velvet	1020	8290.6

considering the remaining 7 datasets, our tool is creating 6.15 times larger contigs compared to Meraculous. It should be noted that all of the input reads in our datasets are 100 bps in length but we define contig's minimum acceptable length as the k value and do not include the actual reads in assembly process and also *N50* calculation, therefore in some of the datasets we obtain *N50* values of less than the actual read sizes.

Comparisons to SOAPdenovo and Velvet show our tool has the best performance in two of the datasets namely datasets 2 and 6 but is behind in other datasets. For datasets from chromosome 10, our tool comes behind the Velvet and SOAPdenovo which can be explained by our different heuristic methods. In two datasets from chromosome 1, our tool outperforms all other assemblers which can again be explained by using multiple k values. It also looks like our tool performs better on small datasets and comes behind the Velvet and SOAPdenovo in large datasets (10M base dataset). Figure 4.1 shows the comparisons in a chart and figure 4.2 shows the largest correct and completely aligned fragment which is built by the assemblers. Fragments are computed by splitting the contigs in locations that have different alignment blocks in their right and left sequences. Based on figure 4.2 our tool has created the best fragment in two of the 9 datasets and falls behind the Velvet and SOAPdenovo for the remaining datasets while always getting better results than Meraculous.

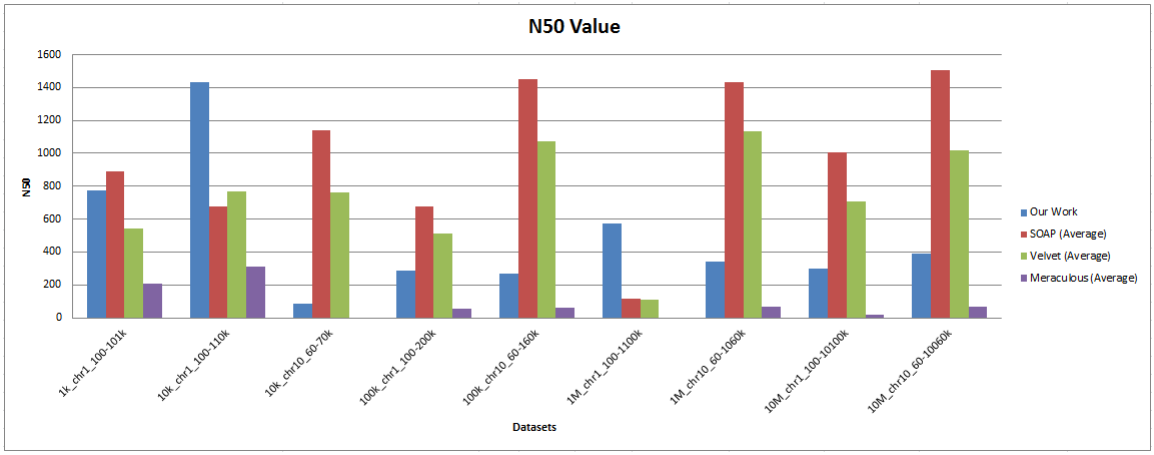

Figure 4.1: N50 results for four assemblers on nine experimented datasets.

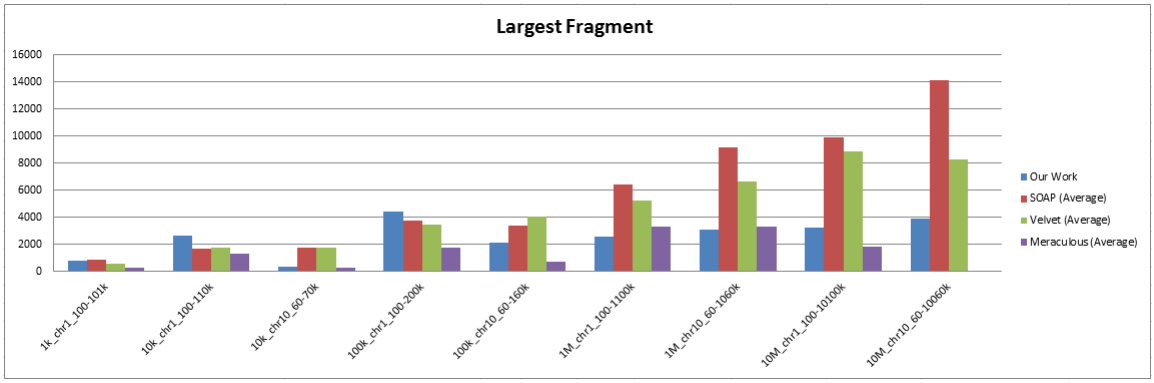

Figure 4.2: Largest fragment produced by four assemblers on nine experimented datasets.

4.2.2 External Contigs Expansion Results

This section presents results for performing the "contigs merging" algorithm described in Algorithm 8 when external contigs from other tools are imported to our system. For each assembler, the best run having the highest *N50* value

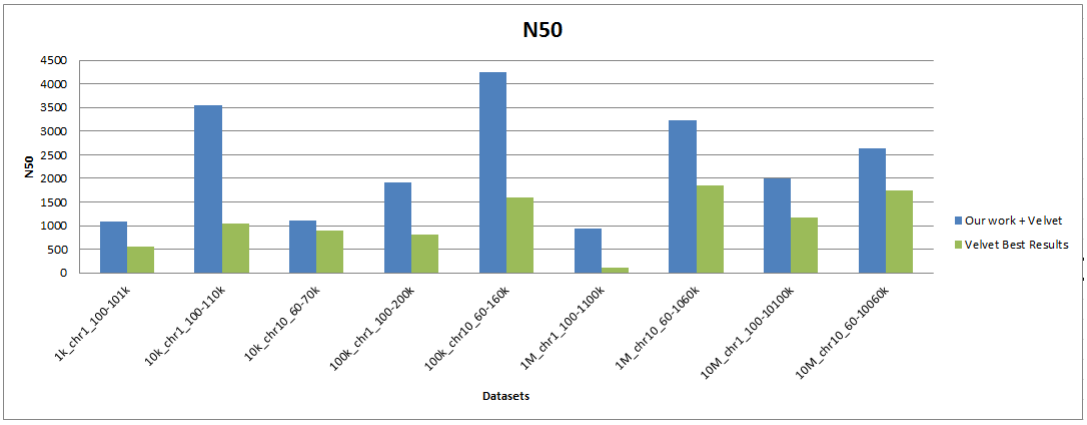

Figure 4.3: Improvements made to Velvet results by combining our tool's result to Velvet contigs. *N50* value is by on avarage a factor of 3.2.

is selected.

Table 4.4 presents experimental results for integration of our tool with the Velvet assembler. Results show that combining our contigs to contigs generated by Velvet significantly increases the quality of results leading to larger fragments from the genome and thus, better *N50* values. All datasets show improvements in results and the *N50* value is increased by on average a factor of 3.2. Figure 4.3 shows the improvements made by this combination for each dataset.

Table 4.5 presents results for integration of our tool with the Meraculous assembler. Results show significant improvement in *N50* values by merging our tool's contigs with contigs generated by Meraculous assembler. The *N50* value is increased by on average a factor of 3.5. In two datasets Meraculous has an *N50* value of zero which means the total sum of all generated contigs'

Table 4.4: Expansion Results for Velvet integration

Assembler	**N50**	**Largest Fragment**
Data Set 1: 1K_Chr1_100-101K		
Velvet	547	547
Our tool + Velvet	1086	1086
Data Set 2: 10K_Chr1_100-110K		
Velvet	1033	2014
Our tool + Velvet	3543	2030
Data Set 3: 10K_Chr10_60-70K		
Velvet	897	1948
Our tool + Velvet	1117	1948
Data Set 4: 100K_Chr1_100-200K		
Velvet	806	2867
Our tool + Velvet	1904	5355
Data Set 5: 100K_Chr10_60-160K		
Velvet	1603	6797
Our tool + Velvet	4255	9014
Data Set 6: 1M_Chr1_100-1100K		
Velvet	104	5322
Our tool + Velvet	936	10230
Data Set 7: 1M_Chr10_60-1060K		
Velvet	1852	11084
Our tool + Velvet	3239	13161
Data Set 8: 10M_Chr1_100-10100K		
Velvet	1167	13690
Our tool + Velvet	1999	18991
Data Set 9: 10M_Chr10_60-10060K		
Velvet	1750	13964
Our tool + Velvet	2632	14018

Table 4.5: Expansion Results for Meraculous integration

Assembler	**N50**	**Largest Fragment**
Data Set 1: 1K_Chr1_100-101K		
Meraculous	324	327
Our tool + Meraculous	772	772
Data Set 2: 10K_Chr1_100-110K		
Meraculous	371	1041
Our tool + Meraculous	1801	2679
Data Set 3: 10K_Chr10_60-70K		
Meraculous	0	355
Our tool + Meraculous	86	355
Data Set 4: 100K_Chr1_100-200K		
Meraculous	84	2157
Our tool + Meraculous	314	4438
Data Set 5: 100K_Chr10_60-160K		
Meraculous	109	1257
Our tool + Meraculous	380	4382
Data Set 6: 1M_Chr1_100-1100K		
Meraculous	0	3333
Our tool + Meraculous	112	3333
Data Set 7: 1M_Chr10_60-1060K		
Meraculous	117	1723
Our tool + Meraculous	360	3077
Data Set 8: 10M_Chr1_100-10100K		
Meraculous	64	3333
Our tool + Meraculous	327	3333
Data Set 9: 10M_Chr10_60-10060K		
Meraculous	120	2172
Our tool + Meraculous	187	4038

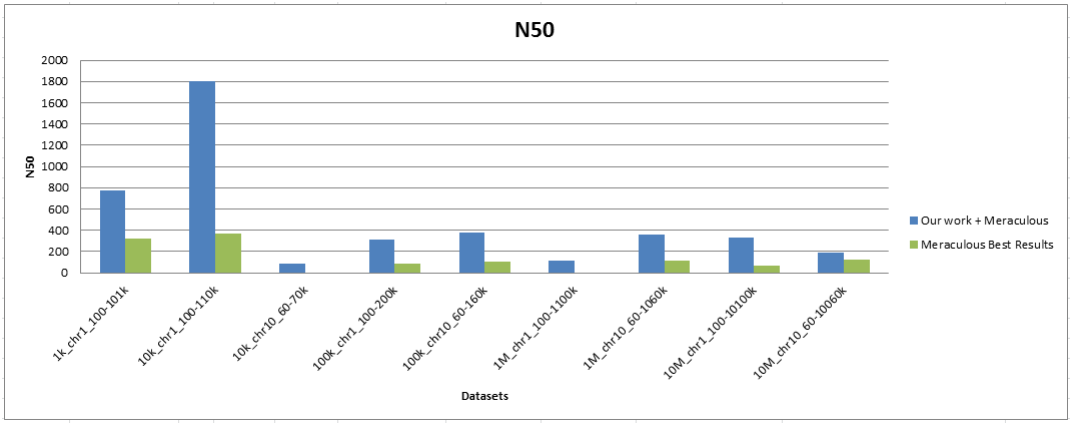

Figure 4.4: Combining our tool's contigs to contigs generated by Meraculous results in significant improvement in assembly results increasing *N50* value by on average a factor of 3.5.

sizes is less than half of the genome length targeted by the dataset, therefore these entries are excluded when calculating the average. Figure 4.4 shows all of the comparisons in a chart.

Table 4.6 presents results for integration of our tool with the SOAPdenovo assembler. Results show that combining our tool's contigs with contigs generated by the SOAPdenovo package also generates better results having larger fragments and *N50* values. The *N50* value is increased by on average a factor of 3.06. Figure 4.5 shows all of the comparisons in a chart.

Results from combining our tool results to outputs from other assemblers supports the idea that it is possible to obtain improved results by merging them to the contigs that are created with different k values in the assembly process. This in fact shows that some of our generated contigs are from the locations that are left over by other assemblers, therefore overlaps can be

Table 4.6: Expansion Results for SOAPdenovo integration

Assembler	**N50**	**Largest Fragment**
Data Set 1: 1K_Chr1_100-101K		
SOAPdenovo	1069	1069
Our tool + SOAPdenovo	1069	1069
Data Set 2: 10K_Chr1_100-110K		
SOAPdenovo	877	1970
Our tool + SOAPdenovo	3527	4169
Data Set 3: 10K_Chr10_60-70K		
SOAPdenovo	1780	1946
Our tool + SOAPdenovo	1946	2353
Data Set 4: 100K_Chr1_100-200K		
SOAPdenovo	1165	5476
Our tool + SOAPdenovo	2346	7994
Data Set 5: 100K_Chr10_60-160K		
SOAPdenovo	2376	5651
Our tool + SOAPdenovo	4550	9460
Data Set 6: 1M_Chr1_100-1100K		
SOAPdenovo	173	10454
Our tool + SOAPdenovo	1677	12823
Data Set 7: 1M_Chr10_60-1060K		
SOAPdenovo	2545	16614
Our tool + SOAPdenovo	4460	19958
Data Set 8: 10M_Chr1_100-10100K		
SOAPdenovo	1790	18684
Our tool + SOAPdenovo	2682	19169
Data Set 9: 10M_Chr10_60-10060K		
SOAPdenovo	2732	27058
Our tool + SOAPdenovo	3761	27089

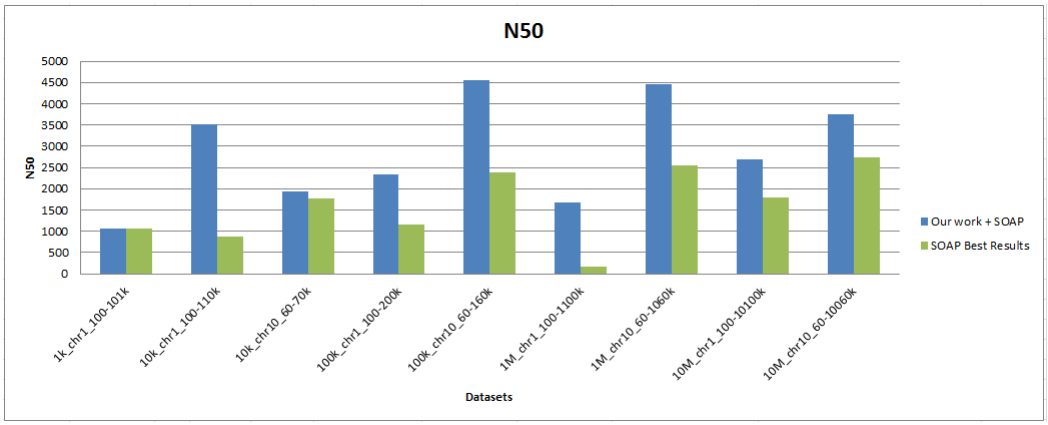

Figure 4.5: Combining our tool's contigs to contigs generated by SOAPdenovo obtain better results having more *N50* values. The *N50* value is increased by on average a factor of 3.06.

found between the results in order to obtain larger fragments. However, there are also false positive links between the merged contigs, thus creating wrong contigs in the results. Currently, we avoid the influence of the false contigs in our results by splitting them from the wrong locations using BLAT and the human reference genome.

4.2.3 Computation Time Results

DNA assemblers usually take a long time to perform especially for large datasets because of loading massive amount of information to memory and processing the information to find overlaps and assemble the fragments. Input data can become massively large making the whole process very slow. Our tool is also not exempted from this fact. Moreover, our tool is designed to

run different k values in parallel which takes more memory usage and requires more time to perform. The contigs merging phase, which is the final phase of our tool, is also time demanding because of using many string matching algorithms on significantly large contigs which is generally considered as a slow process. Figure 4.6 shows detailed information about our tool's run times for different datasets.

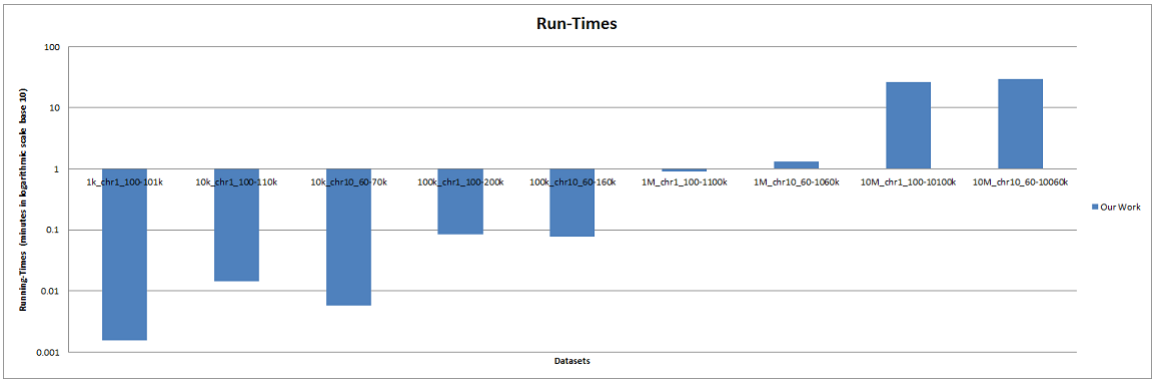

Figure 4.6: Run times of our algorithm for the experimented datasets.

We cannot directly compare our run times to other assemblers, as our tool currently only performs contig creation stage of the assembly process and does not perform the other stages including scaffolding. Other assemblers mainly perform all stages and does not specify a time specifically for contig creation process.

4.3 Summary

This chapter covered our experimental results to investigate our tool's performance on various datasets and compared our results to other assemblers. We chose 9 datasets all extracted from real datasets sequenced by Illumina sequencing technology and we use the human reference gnome version 19 (hg19) in order to align the resulting contigs to the genome and find the accuracy and false links in generated contigs. Results are compared based on the *N50* values and the largest fragments built by the assemblers. We also presented results for the integration of our tool with other assemblers in order to obtain better *N50* values, which is shown to be effective. Experimental results show that our tool can obtain the best results in some datasets based on the repeating patterns and k values selected and is capable of improving the results from other tools by merging them with our generated contigs.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The *de novo* DNA assembly problem is still an open problem to solve, specifically for large genomes including the human genome. This thesis focuses on creating contigs from short reads generated by Next-Generation-Sequencing technology and merging other assemblers' contigs with those generated by our tool in order to obtain improved results. Our algorithm is based on first finding all *k-mers* from the input read set and then filtering the noisy entries by counting the number of occurrences. The *k-mers* are then processed in order to find overlaps, and overlaps that are uniquely expressing on single base-pair extensions are extracted. By having a data structure containing all single base-pair extensions, contigs are created by merging *k-mers* from both directions.

Our algorithm is capable of running the assembly process with several k values in parallel and merging the results from different runs at the end of the assembly. Experimental results show considerable improvements in results when using multiple k values. Our tool is also capable of importing contigs from other assemblers and analyzing them in order to improve results by achieving higher *N50* values.

Our tool is developed in C# and C++ programming langauges and can run in both Windows and Linux machines. It has one main configuration file to load the datasets and assembly parameters. Input files are accepted in .fastq file format and output contigs are generated in .fasta file format. A sample configuration file is presented in figure 3.5 and a sample input .fastq input files is presented in Appendix A.

5.2 Future Work

The *de novo* DNA assembly is a large problem consisting of several parts. Pruning input data sets in order to remove noisy parts, creating contigs based on the short reads, orienting contigs by using mate-pair information and creating scaffolds based on contigs are all different stages of a DNA assembly process. This thesis focuses specifically on creating contigs from short reads, therefore completing other parts in order to have a full *de novo* DNA assembler is a major part of future work.

Finding the correctness of the generated contigs is a difficult problem

in *de novo* assembly because there is no reference genome to compare to. Contigs may be created because of the overlaps that are expressing repeat patterns and not correct extensions. Using pair read information during the contig creation algorithm is one idea that we want to investigate in future. By having the estimated distance between the pair reads in the genome, we want to investigate new ways to create contigs that have less false positive links, leading to more accurate results. Having read pairs can also be useful to generate scaffolds from the contigs. There are also situations that the reference genome is available for the genome and the problem is not *de novo*. We want to add the support for matching the contigs to the available reference genome automatically in order to find their accuracy. This will add a degree of confidence in the accuracy even when applied to *de novo* problems.

Merging results from different assembly runs or external tools generates a number of false positive links between the contigs, leading to having incorrect contigs beside the correct overlaps. This reduces the accuracy of the final results and raises the problem of verifying if the generated final contigs are correct or not. Currently, in our experimental results, we avoid using the wrong contigs to influence the *N50* value by splitting the contigs from the points that left and right alignments are distant as described in chapter 3.5 page 38. New algorithms can be devised as a future work to either detect the false positive links after contigs merging or consider information such as read-pair to reject the false links during the contigs merging algorithm.

Supporting different input file formats is also valuable. Currently only

input files in .fastq file format are supported and all outputs are in .fasta format.

Bibliography

[AIS84] Baruch Awerbuch, Amos Israeli, and Yossi Shiloach. Finding Euler circuits in logarithmic parallel time. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 249–257. ACM, 1984.

[AV84] Mikhail Atallah and Uzi Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29(3):330–337, 1984.

[Bat05] S Batzoglou. Algorithmic challenges in mammalian genome sequence assembly. *Encyclopedia of genomics, proteomics and bioinformatics. John Wiley and Sons*, 2005.

[BFA+] KR Bradnam, JN Fass, A Alexandrov, P Baranay, M Bechner, İ Birol, S Boisvert10, JA Chapman, G Chapuis, R Chikhi, et al. Assemblathon 2: evaluating de novo methods of genome assembly

in three vertebrate species. arXiv e-print 2013. *arXiv preprint arXiv:1301.5406*.

[BJS+02] Serafim Batzoglou, David B Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P Mesirov, and Eric S Lander. ARACHNE: a whole-genome shotgun assembler. *Genome research*, 12(1):177–189, 2002.

[BMK+08] Jonathan Butler, Iain MacCallum, Michael Kleber, Ilya A Shlyakhter, Matthew K Belmonte, Eric S Lander, Chad Nusbaum, and David B Jaffe. ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome research*, 18(5):810–820, 2008.

[BNA+12] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. SPAdes: A new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[CHS+11] Jarrod A Chapman, Isaac Ho, Sirisha Sunkara, Shujun Luo, Gary P Schroth, and Daniel S Rokhsar. Meraculous: de novo genome assembly with short paired-end reads. *PloS one*, 6(8):e23501, 2011.

[CL11] Rayan Chikhi and Dominique Lavenier. Localized genome assembly from reads to scaffolds: practical traversal of the paired string graph. In *Algorithms in Bioinformatics*, pages 39–48. Springer, 2011.

[DLBH07] Juliane C Dohm, Claudio Lottaz, Tatiana Borodina, and Heinz Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome research*, 17(11):1697–1706, 2007.

[DSC+10] Radoje Drmanac, Andrew B Sparks, Matthew J Callow, Aaron L Halpern, Norman L Burns, Bahram G Kermani, Paolo Carnevali, Igor Nazarenko, Geoffrey B Nilsen, George Yeung, et al. Human genome sequencing using unchained base reads on self-assembling DNA nanoarrays. *Science*, 327(5961):78–81, 2010.

[Fle90] Herbert Fleischner. *Eulerian graphs and related topics*, volume 1. Access Online via Elsevier, 1990.

[GJT76] Michael R Garey, David S. Johnson, and R Endre Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM Journal on Computing*, 5(4):704–714, 1976.

[HBB+08] Timothy D Harris, Phillip R Buzby, Hazen Babcock, Eric Beer, Jayson Bowers, Ido Braslavsky, Marie Causey, Jennifer Colonell,

James DiMeo, J William Efcavitch, et al. Single-molecule DNA sequencing of a viral genome. *Science*, 320(5872):106–109, 2008.

[HM99] Xiaoqiu Huang and Anup Madan. CAP3: A DNA sequence assembly program. *Genome research*, 9(9):868–877, 1999.

[HY05] Xiaoqiu Huang and Shiaw-Pyng Yang. Generating a genome assembly with PCAP. *Current Protocols in Bioinformatics*, pages 11–3, 2005.

[JBG+03] David B Jaffe, Jonathan Butler, Sante Gnerre, Evan Mauceli, Kerstin Lindblad-Toh, Jill P Mesirov, Michael C Zody, and Eric S Lander. Whole-genome sequence assembly for mammalian genomes: Arachne 2. *Genome research*, 13(1):91–96, 2003.

[JRB+07] William R Jeck, Josephine A Reinhardt, David A Baltrus, Matthew T Hickenbotham, Vincent Magrini, Elaine R Mardis, Jeffery L Dangl, and Corbin D Jones. Extending assembly of short DNA sequences to handle error. *Bioinformatics*, 23(21):2942–2944, 2007.

[Ken02] W James Kent. BLAT—the BLAST-like alignment tool. *Genome research*, 12(4):656–664, 2002.

[KSS+10] David R Kelley, Michael C Schatz, Steven L Salzberg, et al. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol*, 11(11):R116, 2010.

[Li09] RQ Li. Short Oligonucleotide Analysis Package: SOAPdenovo 1.03. *Beijing Genomics Institute, Beijing*, 2009.

[LLS+11] Yong Lin, Jian Li, Hui Shen, Lei Zhang, Christopher J Papasian, et al. Comparative studies of de novo assembly tools for next-generation sequencing technologies. *Bioinformatics*, 27(15):2031–2037, 2011.

[LZR+10] Ruiqiang Li, Hongmei Zhu, Jue Ruan, Wubin Qian, Xiaodong Fang, Zhongbin Shi, Yingrui Li, Shengting Li, Gao Shan, Karsten Kristiansen, et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome research*, 20(2):265–272, 2010.

[MEA+05] Marcel Margulies, Michael Egholm, William E Altman, Said Attiya, Joel S Bader, Lisa A Bemben, Jan Berka, Michael S Braverman, Yi-Ju Chen, Zhoutao Chen, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005.

[MGG10] Matthew Meyerson, Stacey Gabriel, and Gad Getz. Advances in understanding cancer genomes through second-generation sequencing. *Nature Reviews Genetics*, 11(10):685–696, 2010.

[MKS10] Jason R Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[MPC+09] Kevin Judd McKernan, Heather E Peckham, Gina L Costa, Stephen F McLaughlin, Yutao Fu, Eric F Tsung, Christopher R Clouser, Cisyla Duncan, Jeffrey K Ichikawa, Clarence C Lee, et al. Sequence and structural variation in a human genome uncovered by short-read, massively parallel ligation sequencing using two-base encoding. *Genome research*, 19(9):1527–1541, 2009.

[MPC+11] Paul Medvedev, Son Pham, Mark Chaisson, Glenn Tesler, and Pavel Pevzner. Paired de Bruijn graphs: a novel approach for incorporating mate pair information into genome assemblers. *Journal of Computational Biology*, 18(11):1625–1634, 2011.

[MPC+13] Tanja Magoc, Stephan Pabinger, Stefan Canzar, Xinyue Liu, Qi Su, Daniela Puiu, Luke J Tallon, and Steven L Salzberg. GAGE-B: an evaluation of genome assemblers for bacterial organisms. *Bioinformatics*, 2013.

[MSD+00] Eugene W Myers, Granger G Sutton, Art L Delcher, Ian M Dew, Dan P Fasulo, Michael J Flanigan, Saul A Kravitz, Clark M Mobarry, Knut HJ Reinert, Karin A Remington, et al. A whole-genome assembly of Drosophila. *Science*, 287(5461):2196–2204, 2000.

[Nai07] Achuthsankar S Nair. Computational biology & bioinformatics: a gentle overview. *Communications of Computer Society of India*, 2:1–13, 2007.

[NSW+13] Björn Nystedt, Nathaniel R Street, Anna Wetterbom, Andrea Zuccolo, Yao-Cheng Lin, Douglas G Scofield, Francesco Vezzi, Nicolas Delhomme, Stefania Giacomello, Andrey Alexeyenko, et al. The Norway spruce genome sequence and conifer genome evolution. *Nature*, 2013.

[Pev00] Pavel Pevzner. *Computational molecular biology: an algorithmic approach*, volume 1. MIT press Cambridge, 2000.

[Pop04] Mihai Pop. Shotgun sequence assembly. *Advances in computers*, 60:193–248, 2004.

[Pop09] Mihai Pop. Genome assembly reborn: recent computational challenges. *Briefings in bioinformatics*, 10(4):354–366, 2009.

[PPDS04] Mihai Pop, Adam Phillippy, Arthur L Delcher, and Steven L Salzberg. Comparative genome assembly. *Briefings in bioinformatics*, 5(3):237–248, 2004.

[PTW01] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

[RHR+11] Jonathan M Rothberg, Wolfgang Hinz, Todd M Rearick, Jonathan Schultz, William Mileski, Mel Davey, John H Leamon, Kim Johnson, Mark J Milgrew, Matthew Edwards, et al. An integrated semiconductor device enabling non-optical genome sequencing. *Nature*, 475(7356):348–352, 2011.

[SJ08] Jay Shendure and Hanlee Ji. Next-generation DNA sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[Ske13] Jon Skeet. http://csharpindepth.com/, November 2013.

[SNC77] Frederick Sanger, Steven Nicklen, and Alan R Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 74(12):5463–5467, 1977.

[SWAK95] Granger G Sutton, Owen White, Mark D Adams, and Anthony R Kerlavage. TIGR assembler: A new tool for assembling large shotgun sequencing projects. *Genome Science and Technology*, 1(1):9–19, 1995.

[SWJ+09] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and İnanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[UTK88] Shuichi Ueno, Katsufumi Tsuji, and Yoji Kajitani. On dual Eulerian paths and circuits in plane graphs. In *Circuits and*

Systems, 1988., IEEE International Symposium on, pages 1835–1838. IEEE, 1988.

[WC+53] James D Watson, Francis HC Crick, et al. Molecular structure of nucleic acids. *Nature*, 171(4356):737–738, 1953.

[WSJH07] René L Warren, Granger G Sutton, Steven JM Jones, and Robert A Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, 2007.

[ZB08] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008.

Appendices

Appendix A

Sample .FastQ file which is the input to our assembly tool. First line contains the ID for the sequence, second line is the actual sequence, third line contains only a plus character and the fourth line contains the quality scores for every base-pair of the sequence.

```
1  FCD19T7ACXX:1:2313:16262:81103#
   AACACGGACAGCTCCCTGAACTCCAGGAAACATCCTGATTTAGTGTTTTGAGTATTGT
3  GAAGCACAGTTAGAGCAGAAACATGGAGAATCACCTTAAATG
   +
5  _bbeeeeegfggfhiiiihhhiidhhhihbgdffhii^agffhac^efhfZebgggif
   hiihhhhcddgdf__a^_dd]bbdcbb_`acccccccbb_bd
7  FCD19T7ACXX:1:1105:2330:89450#
   TAATGTCTAGAATCTGAGTGCCATGTTATCAAATTGTACTGAGACTCTTGCAGTCACA
9  CAGGCTGACATGTAAGCATCGCCATGCCTAGTACAGACTCTC
   +
11 ____eceeecggggdfh[ebghfdhffffhcgdggfS[bffaafghecgfaefghhh`_
   efdgeffZbbfgeZegfcdgfgggecbcdbddcd`b_abbbb
13 FCC0YLPACXX:1:2114:18570:19681#
   AGCACACAGAGAATAATGTCTAGAATCTGAGTGCCATATTATCAAATTGTACTGAGAC
15 TCTTGCAGTCACACAGGCTGACATGTAAGCATCGCCATGCCT
   +
17 __beeeeeggggiiiiiiiiiiiiiiihighiiiihhiiihiiiiiihiihifgce
   ghiiighihiihiihhfddgggfggeededeedcccccccc
19 FCC0YLPACXX:2:1209:18161:19463#
   CATGTTATCAAATTGTACTGAGACTCTTGCAGTCACACAGGCTGACATGTAAGCATCG
21 CCATGCCTAGTACAGACTCTCCCTGCAGATGAAATTATATGG
```

```
+
bb_eceeecggfghidghhhffdghiiiiiiic‘ffbghichifghiiiiifg]agff
gfhhiihihh_dfffcgbbdeebeecbdcbdccccbbccccc
FCC0YLPACXX:2:1111:2648:57787#
TGTGTTGCTGAGAACTGCTCAGTAACACGGACAGCTCCCTGAACTCCAGGAAACATCC
TGATTTAGTGTTTTGAGTATTGTGAAGCACAGTTAGAGCAGA
+
```

Appendix B

Main class definitions and codes are described in this appendix.

- K-mer Class

```
public class Kmer {
  public List<byte> PrimaryLeftExtensions = new List<byte>()
    ;
    public List<byte> FinalRightExtensions = new List<byte
   >();
    public List<byte> FinalLeftExtensions = new List<byte>()
   ;
    public byte[] RightCounts = new byte[4] { 0, 0, 0, 0 };
   //0:A, 1: C, 2: G, 3: T
  public byte[] LeftCounts = new byte[4] { 0, 0, 0, 0 };
   //0: A, 1: C, 2: G, 3: T
    public byte RepeatsInReads;
```

```
    public ExtensionLabel LeftExtension;
    public ExtensionLabel RightExtension;

    public Kmer() {}
}

public enum ExtensionLabel {
  DEAD_END = 0,
  RESOLVED = 1,
  MAJORITY_VOTED = 2,
  UNRESOLVED = 3
}

public enum KmerOverlapDirection {
  L = 0,
  R = 1
}
```

- Read Class

```
public class Read {
  private byte[] seq;
  private byte[] qualityString = new byte[100];
  public byte[] Seq {
    get { return seq; }
        set { seq = value; }
  }
```

```
8   public byte[] QualityString {
      get { return qualityString; }
10    set { qualityString = value; }
    }
12    public int GetQualityValue(int index) {
          return ((int) qualityString[index]) - Parameters.
     BaseQualityValue;
14  }
    public Read(byte[] seq, int pair_idx) {
16    this.seq = seq;
    }
18  }
```

- IOHandler Class: Responsible for parsing/loading input data and configuration file

```
public class IOHandler
2   {
        #region Public Methods
4
        public IOHandler(AssemblyCore assemblyCore)
6       {
            this.assemblyCore = assemblyCore;
8       }

10      private static string ReadXMLValue(XmlNodeList nodes,
       string key)
```

```
    {
12          foreach (XmlNode node in nodes)
            {
14              if (node.Attributes["key"].Value.Equals(key))
                    return node.Attributes["value"].Value;
16          }
            return string.Empty;
18      }


20

        public static void LoadConfigurationFile(string
        fileAddress)
22      {
            XmlDocument doc = new XmlDocument();
24          doc.Load(fileAddress);
            XmlNodeList nodes = doc.SelectNodes("
        AssemblyParameters/AssemblyParameter");
26

            Parameters.ReadSet1Address = ReadXMLValue(nodes, "
        ReadSet1Address");
28          Parameters.ReadSet2Address = ReadXMLValue(nodes, "
        ReadSet2Address");
            Parameters.ReferenceGenomeAddress = ReadXMLValue(
        nodes, "RefrenceGenomeAddress");
30          /*if (Parameters.ReferenceGenomeAddress != string.
        Empty)
                IOHandler.ReadReferenceGenome();*/
```

```
32          Parameters.ContigsOutputAddress = ReadXMLValue(nodes
    , "ContigsOutputAddress");

34          Parameters.ConsiderPairReads = bool.Parse(
    ReadXMLValue(nodes, "ConsiderPairReads"));
            Parameters.multipleKs = bool.Parse(ReadXMLValue(
    nodes, "MultipleKs"));
36
            string KValues = ReadXMLValue(nodes, "Ks");
38          string[] splits = KValues.Split(',');
            foreach (string k in splits)
40              Parameters.KList.Add(int.Parse(k));
            Parameters.ContigsMinLength = int.Parse(ReadXMLValue
    (nodes, "ContigsMinLength"));
42
            Parameters.MultiplicityMinThreshold = int.Parse(
    ReadXMLValue(nodes, "MultiplicityMinThreshold"));
44          Parameters.HighQualityMinThreshold = int.Parse(
    ReadXMLValue(nodes, "HighQualityMinThreshold"));
            Parameters.ContigsOverlapValue = int.Parse(
    ReadXMLValue(nodes, "ContigsOverlapValue"));
46
            Parameters.ResolveNotUUExtensions = bool.Parse(
    ReadXMLValue(nodes, "ResolveNotUUExtensions"));
48          Parameters.MajorityVotingThreshold = double.Parse(
    ReadXMLValue(nodes, "MajorityVotingThreshold"));
```

```
50          Parameters.BaseQualityValue = int.Parse(ReadXMLValue
    (nodes, "BaseQualityValue"));

52          Console.WriteLine("Configurations are loaded...");
    }
54

56      public static void LoadReads()
    {
58          StreamReader sr1 = null, sr2 = null;
            sr1 = new StreamReader(Parameters.ReadSet1Address);
60          if (Parameters.ConsiderPairReads)
                sr2 = new StreamReader(Parameters.
    ReadSet2Address);
62
            int lidx = 2;
64          int idx = 0;
            while (!sr1.EndOfStream)
66          {
                sr1.ReadLine();
68              string r1_seq = sr1.ReadLine();
                sr1.ReadLine();
70              string r1_qual = sr1.ReadLine();

72              string r2_seq = string.Empty;
                string r2_qual = string.Empty;
74
```

```
            if (Parameters.ConsiderPairReads)
76          {
                sr2.ReadLine();
78              r2_seq = sr2.ReadLine();
                sr2.ReadLine();
80              r2_qual = sr2.ReadLine();
            }
82
            Read r1, r1rc;
84          Read r2, r2rc;

86          byte[] r1_seq_bytes = Utilities.BitEncode(r1_seq
    );


88


90          r1 = new Read(r1_seq_bytes, 0);
            for (int i = 0; i < r1_qual.Length; i++)
92              r1.QualityString[i] = (byte) r1_qual[i];

94          string r1_seq_rc = Utilities.GetRC(r1_seq);
            byte[] r1_seq_rc_bytes = Utilities.BitEncode(
    r1_seq_rc);
96          r1rc = new Read(r1_seq_rc_bytes, 0);
            Array.Copy(r1.QualityString, 0, r1rc.
    QualityString, 0, r1.QualityString.Length);
98          Array.Reverse(r1rc.QualityString);
```

```
100             if (Parameters.ConsiderPairReads)
                {
102                 byte[] r2_seq_bytes = Utilities.BitEncode(
    r2_seq);
                    r2 = new Read(r2_seq_bytes, 1);
104                 for (int i = 0; i < r2_qual.Length; i++)
                        r2.QualityString[i] = (byte) r2_qual[i];
106
                    string r2_seq_rc = Utilities.GetRC(r2_seq);
108                 byte[] r2_seq_rc_bytes = Utilities.BitEncode
    (r2_seq_rc);

110                 r2rc = new Read(r2_seq_rc_bytes, 0);
                    Array.Copy(r2.QualityString, 0, r2rc.
    QualityString, 0, r2.QualityString.Length);
112                 Array.Reverse(r2rc.QualityString);

114                 AssemblyGlobal.Reads.Add(r1);
                    AssemblyGlobal.Reads.Add(r1rc);
116
                    AssemblyGlobal.Reads.Add(r2);
118                 AssemblyGlobal.Reads.Add(r2rc);
                }
120             else
                {
122
```

```
                AssemblyGlobal.Reads.Add(r1);
124
                AssemblyGlobal.Reads.Add(r1rc);
126             }
            }
128         sr1.Close();
            if (Parameters.ConsiderPairReads)
130             sr2.Close();
        }
132
        public static void ReadReferenceGenome()
134     {
            if (AssemblyGlobal.ReferenceGenomeData != string.
    Empty)
136             return;//this "static" method must be called
    only once

138         StreamReader sr = new StreamReader(Parameters.
    ReferenceGenomeAddress);
            string genome = string.Empty;
140         while (!sr.EndOfStream)
                genome += sr.ReadLine();
142         AssemblyGlobal.ReferenceGenomeData = genome;
            Parameters.RefGenomeAvailable = true;
144         sr.Close();
        }
146
```

```
    public void PrintContigs()
148 {
        int K = assemblyCore.K;
150     StreamWriter sw = new StreamWriter(Parameters.
    ContigsOutputAddress);
        int idx = 0;
152     assemblyCore.contigs.Sort(delegate(UUContig c1,
    UUContig c2)
        {
154         if (c1.Sequence.Length <= c2.Sequence.Length)
                return 1;
156         return -1;
        });
158     StringBuilder sb = new StringBuilder();
        List<UUContig> wrongToBeRemoved = new List<UUContig
    >();
160     List<UUContig> smallToBeRemoved = new List<UUContig
    >();
        foreach (UUContig contig in assemblyCore.contigs)
162     {
            //string contigStr = ASCIIEncoding.ASCII.
    GetString(contig.Sequence.ToArray());
164         string contigStr = Utilities.BitDecode(contig.
    Sequence, contig.OccupiedCellCounts, contig.LastPos);
            if (contigStr.Length <= 2 * K)
166         {
                smallToBeRemoved.Add(contig);
```

```
168                 continue;
            }
170             bool foundInGenome = false;
            if (Parameters.RefGenomeAvailable)
172             {
                if (AssemblyGlobal.ReferenceGenomeData.
    Contains(contigStr) || AssemblyGlobal.
    ReferenceGenomeData.Contains(ASCIIEncoding.ASCII.
    GetString(Utilities.GetRC(contig.Sequence).ToArray())))
174                 foundInGenome = true;
                else
176             {
                    foundInGenome = false;
178                 wrongToBeRemoved.Add(contig);
                }
180         }

182         /*sb.AppendLine(string.Format("id:{0} size:{1}
    quality: {2}found:{3} seq:{4}", idx, contig.Sequence.
    Length,
                                                contig.Quality,
    foundInGenome ? "yes" : "no", contig.Sequence));*/
184         sb.AppendLine(">Contig_" + idx.ToString() + "
    _Size_" + contig.Sequence.Count().ToString() + "
    _LeftChar:" +
                                    contig.LeftExtensionChar + "
    _RightChar:" + contig.RightExtensionChar);
```

```
186                 sb.AppendLine(contigStr);
                    idx++;
188             }
                sw.Write(sb.ToString());
190             sw.Close();

192             /*foreach (UUContig cntg in wrongToBeRemoved)
                {
194                 AssemblyCore.contigs.Remove(cntg);
                }*/
196
                foreach (UUContig cntg in smallToBeRemoved)
198             {
                    assemblyCore.contigs.Remove(cntg);
200             }
        }
202
        #endregion
204
        #region Private Methods
206
        private AssemblyCore assemblyCore;
208
        #endregion
210
}
```

- Parameters Class holding values for all settings/configuration of the assembler

```
public class Parameters
{
    public static string ReadSet1Address;
    public static string ReadSet2Address;
    public static string ReferenceGenomeAddress;
    public static string ContigsOutputAddress;
    public static bool RefGenomeAvailable;




    public static bool ConsiderPairReads = true;
    public static bool multipleKs;

    //public static int K;
    public static List<int> KList = new List<int>();
    public static int ContigsMinLength;
    public static int MultiplicityMinThreshold;
    public static int HighQualityMinThreshold;
    public static int ContigsOverlapValue;
    public static double MajorityVotingThreshold;

    public static bool ResolveNotUUExtensions;

    public static int BaseQualityValue;

```

```
    public static bool ParallelRun;
27  public static bool DoLog;
    public static string ConfigFileAddress;
29  public static bool LoadExtCntgs;
    public static string ExternalContigFileAddress;
31

}
```

- Utilities class containing bit Encoder/Decoder algorithms

```
public static class Utilities
2 {


4   public static int GetDNABPIndex(byte BP)
    {
6       switch ((char)BP)
        {
8           case 'A':
                return 0;
10          case 'C':
                return 1;
12          case 'G':
                return 2;
14          case 'T':
                return 3;
16      }
        return -1;
```

```
18      }

20      public static string GetRC(string seq)
        {
22          StringBuilder sb = new StringBuilder(seq);
            for (int i = seq.Length - 1; i >= 0; i--)
24              sb[seq.Length - 1 - i] = GetComplementBP(seq[i])
    ;
            return sb.ToString();
26      }

28      public static ByteArrayWrapper GetRC(ByteArrayWrapper
    seq)
        {
30          return new ByteArrayWrapper(GetRC(seq.Bytes));
        }
32
        public static byte[] GetRC(byte[] seq)
34      {
            byte[] result = new byte[seq.Length];
36          for (int i = seq.Length-1; i>=0; i--)
            {
38              result[seq.Length - 1 - i] = GetComplementBP(seq
    [i]);
            }
40          return result;
        }
```

```
42
    public static List<byte> GetRC(List<byte> seq)
44  {
        List<byte> result = new List<byte>();
46      for (int i = seq.Count - 1; i >= 0; i--)
        {
48          result.Add(GetComplementBP(seq[i]));
        }
50      return result;
    }
52
    public static string GetReverse(string seq)
54  {
        char[] arr = seq.ToCharArray();
56      Array.Reverse(arr);
        return new string(arr);
58  }

60  // Copyright (c) 2008-2013 Hafthor Stefansson
    // Distributed under the MIT/X11 software license
62  // Ref: http://www.opensource.org/licenses/mit-license.
    php.
    public static unsafe bool UnsafeCompare(byte[] a1, byte
    [] a2)
64  {
        if (a1 == null || a2 == null || a1.Length != a2.
    Length)
```

```
66              return false;
        fixed (byte* p1 = a1, p2 = a2)
68          {
                byte* x1 = p1, x2 = p2;
70              int l = a1.Length;
                for (int i = 0; i < l/8; i++, x1 += 8, x2 += 8)
72                  if (*((long*) x1) != *((long*) x2)) return
    false;
                if ((l & 4) != 0)
74              {
                    if (*((int*) x1) != *((int*) x2)) return
    false;
76                  x1 += 4;
                    x2 += 4;
78              }
                if ((l & 2) != 0)
80              {
                    if (*((short*) x1) != *((short*) x2)) return
     false;
82                  x1 += 2;
                    x2 += 2;
84              }
                if ((l & 1) != 0) if (*((byte*) x1) != *((byte*)
    x2)) return false;
86              return true;
            }
88      }
```

```
90      public static unsafe string GetReverseUnsafe(string seq)
        {
92          //assuming the seq size is usually 100 the fastest
        method is unsafe pointer reverse... (http://charter.
        herokuapp.com/MZ02Y32T/performance-of-selected-string-
        reversal-methods-lower-is-better)
            int len = seq.Length;
94
            // Why allocate a char[] array on the heap when you
        won't use it
96          // outside of this method? Use the stack.
            char* reversed = stackalloc char[len];
98
            // Avoid bounds-checking performance penalties.
100         fixed (char* str = seq)
            {
102             int i = 0;
                int j = i + len - 1;
104             while (i < len)
                {
106                 reversed[i++] = str[j--];
                }
108         }

110         // Need to use this overload for the System.String
        constructor
```

```
        // as providing just the char* pointer could result
    in garbage
112         // at the end of the string (no guarantee of null
    terminator).
        return new string(reversed, 0, len);
114     }

116     public static byte GetComplementBP(byte bp)
    {
118         switch(bp)
        {
120             case (byte)'A':
                    return (byte) 'T';
122             case (byte)'T':
                    return (byte)'A';
124             case (byte)'C':
                    return (byte)'G';
126             case (byte)'G':
                    return (byte)'C';
128         }
        return (byte)'X';
130     }

132     public static char GetComplementBP(char bp)
    {
134         switch (bp)
        {
```

```
136             case 'A':
                    return 'T';
138             case 'T':
                    return 'A';
140             case 'C':
                    return 'G';
142             case 'G':
                    return 'C';
144         }
            return 'X';
146     }

148     public static int CalculateN50Statistic(int genomeSize,
        List<int> orderedContigSizes)
        {
150         int total = 0;
            int genomeHalfSize = genomeSize / 2;
152         foreach (int size in orderedContigSizes)
            {
154             total += size;
                if (total >= genomeHalfSize)
156                 return size;
            }
158         return -1;
        }
160

        public static byte GetBPByIndex(byte[] src, int index,
```

```
    int originalSrcLength)
162 {
        int cellIndex = index/4;
164     int pos = (index%4)*2;
        byte b = src[cellIndex];
166     switch (pos)
        {
168         case 0:
                b = (byte) (b & 3);
170             break;
            case 2:
172             b = (byte) ((b & 12) >> 2);
                break;
174         case 4:
                b = (byte) ((b & 48) >> 4);
176             break;
            case 6:
178             b = (byte) ((b & 192) >> 6);
                break;
180     }
        switch (b)
182     {
            case 0:
184             return (byte)'A';
            case 1:
186             return (byte)'C';
            case 2:
```

```
188                 return (byte)'G';
            case 3:
190                 return (byte)'T';
        }
192     return (byte)'Z';
    }
194
    /*public static void ChangeBP(ref byte[] dna, byte newBP
    , int indexInString)
196 {
        int cellIndex = indexInString/4;
198     int posInCell = (indexInString%4)*2;

200     byte r = 0;
        switch (newBP)
202     {
            case (byte) 'A':
204             r = 0;
                break;
206         case (byte) 'C':
                r = 1;
208             break;
            case (byte) 'G':
210             r = 2;
                break;
212         case (byte) 'T':
                r = 3;
```

```
214                 break;
        }
216
        r = (byte) (r << posInCell);
218     byte b = dna[cellIndex];
        switch (posInCell)
220     {
            case 0:
222             b = (byte) (b & 252); //b & 11111100 => b:
    XXXXXX00
                b = (byte) (b | r);
224             break;
            case 1:
226             b = (byte) (b & 243); //b & 11110011 => b:
    XXXX00XX
                b = (byte) (b | r);
228             break;
            case 2:
230             b = (byte) (b & 207); //b & 11001111 => b:
    XX00XXXX
                b = (byte) (b | r);
232             break;
            case 3:
234             b = (byte) (b & 63); //b & 00111111 => b:00
    XXXXXX
                b = (byte) (b | r);
236             break;
```

```
        }
238         dna[cellIndex] = b;
    }*/
240
    public static byte[] BitEncode(byte[] str)
242 {
        return BitEncode(ASCIIEncoding.ASCII.GetString(str))
    ;
244 }

246 public static byte[] BitEncode(string str)
    {
248     // A: 00, C: 01 , G: 11, D: 10 MSB -> LSB
        //each character is encoded by 2 bits as above...
250     byte[] b = new byte[(int)Math.Ceiling(str.Length /
    4.0)];
        byte cellIndex = 0;
252     byte pos = 0;
        foreach (char c in str)
254     {
            byte r = 0;
256         if (c == 'A')
                r = 0; //00000000
258         else if (c == 'C')
                r = 1; //00000001
260         else if (c == 'G')
                r = 2; //00000010
```

```
262             else if (c == 'T')
                    r = 3; //00000011
264
                r = (byte) (r << pos);
266             b[cellIndex] = (byte) (b[cellIndex] | r);
                if (pos == 6)
268             {
                    cellIndex++;
                    /*if (b.Length == cellIndex)
270                     Array.Resize(ref b, b.Length + 5);*/
                }
272             pos = (byte) ((pos + 2)%8);
            }
274         return b;
        }

278     public static void BitCopyArray(byte[] src, int
    srcStartIdx, ref byte[] dest, int destStartIdx, int
    length)
        {
280         int srcPos = (srcStartIdx%4)*2;
            int srcCell = srcStartIdx/4;
282
            int destPos = (destStartIdx%4)*2;
284         int destCell = destStartIdx/4;

286         int endPos = ((srcStartIdx + length)%4)*2;
```

```
        int endCell = (srcStartIdx + length)/4;
288
        while (true)
        {
290         if (srcCell == endCell && srcPos == endPos)
                break;

294         byte srcByte = src[srcCell];
            byte c = 0;
296         switch (srcPos)
            {
298             case 0:
                    c = (byte) (srcByte & 3);
300                 break;
                case 2:
302                 c = (byte) ((srcByte & 12) >> 2);
                    break;
304             case 4:
                    c = (byte) ((srcByte & 48) >> 4);
306                 break;
                case 6:
308                 c = (byte) ((srcByte & 192) >> 6);
                    break;
310         }
            //now c is our BP... either A (00), C(01), G(10)
    , or T(11)
312
```

```
            c = (byte) (c << destPos);
314         byte destByte = dest[destCell];
            switch (destPos)
316         {
                case 0:
318                 destByte = (byte) (destByte & 252); //b
    & 11111100 => b:XXXXXX00
                    destByte = (byte) (destByte | c);
320                 break;
                case 2:
322                 destByte = (byte) (destByte & 243); //b
    & 11110011 => b:XXXX00XX
                    destByte = (byte) (destByte | c);
324                 break;
                case 4:
326                 destByte = (byte) (destByte & 207); //b
    & 11001111 => b:XX00XXXX
                    destByte = (byte) (destByte | c);
328                 break;
                case 6:
330                 destByte = (byte) (destByte & 63); //b &
     00111111 => b:00XXXXXX
                    destByte = (byte) (destByte | c);
332                 break;
            }
334         dest[destCell] = destByte;
```

```
336             srcPos = (srcPos + 2)%8;
                if (srcPos == 0)
338                 srcCell++;

340             destPos = (destPos + 2)%8;
                if (destPos == 0)
342                 destCell++;
            }
344     }

346     public static string BitDecode(byte[] b, int
    occupiedCellsCount, int lastPos)
        {
348         int t = 0;
            int maxT = (occupiedCellsCount - 1)*4 + lastPos/2;
350         string str = string.Empty;
            foreach (byte b1 in b)
352         {
                byte[] masks = new byte[4];
354             masks[0] = (byte) (b1 & 3);
                masks[1] = (byte) ((b1 & 12) >> 2);
356             masks[2] = (byte) ((b1 & 48) >> 4);
                masks[3] = (byte) ((b1 & 192) >> 6);
358
                foreach (byte mask in masks)
360             {
                    if (t == maxT)
```

```
362                     break;
                    if (mask == 0)
364                     str += 'A';
                    else if (mask == 1)
366                     str += 'C';
                    else if (mask == 2)
368                     str += 'G';
                    else if (mask == 3)
370                     str += 'T';
                    t++;
372             }
        }
374     return str;
    }
376
    public static bool BitCompare(byte[] a, int startIndex_a
    , byte[] b, int startIndex_b, int length)
378 {
        int a_pos = (startIndex_a%4)*2;
380     int a_cell = startIndex_a/4;

382     int b_pos = (startIndex_b%4)*2;
        int b_cell = startIndex_b/4;
384
        int idx = 0;
386     while (idx < length)
        {
```

```
388             byte a_byte = a[a_cell];
                byte b_byte = b[b_cell];
390             byte a_bp = 0;
                switch (a_pos)
392             {
                    case 0:
394                     a_bp = (byte) (a_byte & 3);
                        break;
396                 case 2:
                        a_bp = (byte) ((a_byte & 12) >> 2);
398                     break;
                    case 4:
400                     a_bp = (byte) ((a_byte & 48) >> 4);
                        break;
402                 case 6:
                        a_bp = (byte) ((a_byte & 192) >> 6);
404                     break;
                }
406             //a_bp is the BP now... either A (00), C(01), G
    (10), or T(11)
                byte b_bp = 0;
408             switch (b_pos)
                {
410                 case 0:
                        b_bp = (byte) (b_byte & 3);
412                     break;
                    case 2:
```

```
414                     b_bp = (byte) ((b_byte & 12) >> 2);
                        break;
416                 case 4:
                        b_bp = (byte) ((b_byte & 48) >> 4);
418                     break;
                    case 6:
420                     b_bp = (byte) ((b_byte & 192) >> 6);
                        break;
422             }

424             if (a_bp != b_bp)
                    return false;
426             if (a_pos == 6)
                {
428                 a_pos = 0;
                    a_cell++;
430             }
                else
432                 a_pos += 2;

434             if (b_pos == 6)
                {
436                 b_pos = 0;
                    b_cell++;
438             }
                else
440                 b_pos += 2;
```

```
            idx++;
        }
        return true;
    }

    public static byte[] BitRC(byte[] src, int
occupiedCellCounts, int lastPos)
    {
        byte[] result = new byte[src.Length];

        int srcPos;
        if (lastPos == 0)
            srcPos = 6;
        else
            srcPos = lastPos - 2;
        int srcCell = occupiedCellCounts - 1;

        int resultPos = 0;
        int resultCell = 0;

        while (true)
        {
            byte srcByte = src[srcCell];
            byte c = 0;
            switch (srcPos)
            {
                case 0:
```

```
                    c = (byte) (srcByte & 3);
468                 break;
                case 2:
470                 c = (byte) ((srcByte & 12) >> 2);
                    break;
472             case 4:
                    c = (byte) ((srcByte & 48) >> 4);
474                 break;
                case 6:
476                 c = (byte) ((srcByte & 192) >> 6);
                    break;
478         }
            //now c is our BP... either A (00), C(01), G(10)
    , or T(11)
480         //get complement of c
            switch (c)
482         {
                case 0: //A
484                 c = 3;
                    break;
486             case 1: //C
                    c = 2;
488                 break;
                case 2: //G
490                 c = 1;
                    break;
492             case 3: //T
```

```
                    c = 0;
494                 break;
            }
496
            c = (byte) (c << resultPos);
498         byte destByte = result[resultCell];
            switch (resultPos)
500         {
                case 0:
502                 destByte = (byte) (destByte & 252); //b
    & 11111100 => b:XXXXXX00
                    destByte = (byte) (destByte | c);
504                 break;
                case 2:
506                 destByte = (byte) (destByte & 243); //b
    & 11110011 => b:XXXX00XX
                    destByte = (byte) (destByte | c);
508                 break;
                case 4:
510                 destByte = (byte) (destByte & 207); //b
    & 11001111 => b:XX00XXXX
                    destByte = (byte) (destByte | c);
512                 break;
                case 6:
514                 destByte = (byte) (destByte & 63); //b &
     00111111 => b:00XXXXXX
                    destByte = (byte) (destByte | c);
```

```
516                     break;
                }
518             result[resultCell] = destByte;

520             if (srcCell == 0 && srcPos == 0)
                    break;
522             if (srcPos == 0)
                {
524                 srcPos = 6;
                    srcCell--;
526             }
                else
528                 srcPos -= 2;
                if (resultPos == 6)
530             {
                    resultPos = 0;
532                 resultCell++;
                }
534             else
                    resultPos+=2;
536         }
            return result;
538     }
}
```

- ByteArrayComparer class used for generating hash codes and equality

methods for byte arrays in C-Sharp

```
public class ByteArrayComparer : IEqualityComparer<byte[]>
{
    public bool Equals(byte[] a1, byte[] a2)
    {
        if (a1.Length != a2.Length)
            return false;
        for (int i = 0; i < a1.Length; i++)
            if (a1[i] != a2[i])
                return false;
        return true;
    }

    public static bool StaticEquals(byte[] a1, byte[] a2)
    {
        for (int i = 0; i < a1.Length; i++)
            if (a1[i] != a2[i])
                return false;
        return true;
    }

    public int GetHashCode(byte[] str)
    {
        unchecked
        {
            const int p = 16777619;
            int hash = (int) 2166136261;
```

```
27
            for (int i = 0; i < str.Length; i++)
29              hash = (hash ^ str[i])*p;

31          hash += hash << 13;
            hash ^= hash >> 7;
33          hash += hash << 3;
            hash ^= hash >> 17;
35          hash += hash << 5;
            return hash;
37      }
    }
39 }
```

- AssemblyGlobal class containing global containers for K-mer and Read objects and creating CoreAssembly objects for assembly runs.

```
1 public static class AssemblyGlobal
  {
3     public static string ReferenceGenomeData = string.Empty;
      public static List<UUContig> contigs = new List<UUContig
    >();
5     public static List<Read> Reads = new List<Read>();

7     private static Dictionary<Thread, AssemblyCore>
    ThreadsToAssemblyCores = new Dictionary<Thread,
    AssemblyCore>();
```

```
9       private static void PrintContigs()
        {
11          StreamWriter sw = new StreamWriter(Parameters.
        ContigsOutputAddress);
            int idx = 0;
13          contigs.Sort(delegate(UUContig c1, UUContig c2)
            {
15              if (c1.Sequence.Length <= c2.Sequence.Length)
                    return 1;
17              return -1;
            });
19          StringBuilder sb = new StringBuilder();
            List<UUContig> wrongToBeRemoved = new List<UUContig
        >();
21          List<UUContig> smallToBeRemoved = new List<UUContig
        >();
            foreach (UUContig contig in contigs)
23          {
                //string contigStr = ASCIIEncoding.ASCII.
        GetString(contig.Sequence.ToArray());
25              string contigStr = Utilities.BitDecode(contig.
        Sequence, contig.OccupiedCellCounts, contig.LastPos);
                if (contigStr.Length <= Parameters.
        ContigsMinLength)
27              {
                    smallToBeRemoved.Add(contig);
```

```
29                continue;
            }
31          bool foundInGenome = false;
            if (Parameters.RefGenomeAvailable)
33          {
                if (AssemblyGlobal.ReferenceGenomeData.
Contains(contigStr) || AssemblyGlobal.
ReferenceGenomeData.Contains(ASCIIEncoding.ASCII.
GetString(Utilities.GetRC(contig.Sequence).ToArray())))
35                  foundInGenome = true;
                else
37              {
                    foundInGenome = false;
39                  wrongToBeRemoved.Add(contig);
                }
41          }

43          /*sb.AppendLine(string.Format("id:{0} size:{1}
quality: {2}found:{3} seq:{4}", idx, contig.Sequence.
Length,
                                            contig.Quality,
foundInGenome ? "yes" : "no", contig.Sequence));*/
45          sb.AppendLine(">Contig_" + idx.ToString() + "
_Size_" + contig.Sequence.Count().ToString() + "
_LeftChar:" +
                                contig.LeftExtensionChar + "
_RightChar:" + contig.RightExtensionChar);
```

```
47              sb.AppendLine(contigStr);
                idx++;
49          }
            sw.Write(sb.ToString());
51          sw.Close();
            foreach (UUContig contig in smallToBeRemoved)
53          {
                contigs.Remove(contig);
55          }

57      }

59      public static void Log(string msg)
        {
61          if (Parameters.DoLog)
                Console.WriteLine(msg);
63      }

65      public static void DoAssembly()
        {
67          IOHandler.LoadConfigurationFile(Parameters.
        ConfigFileAddress);

69          IOHandler.LoadReads();

71          if (Parameters.ParallelRun)
            {
```

```
73              Log("Parallel run...");
                List<Task> assemblyTasks = new List<Task>();
75              for (int i = 0 ; i < Parameters.KList.Count; i
    ++)
                {
77                  int kTemp = Parameters.KList[i];
                    Log("Task K = " + kTemp + " added...");
79                  AssemblyCore core = new AssemblyCore(kTemp);
                    assemblyTasks.Add(Task.Factory.StartNew(core
    .DoThreadedAssembly));
81              }
                Task.WaitAll(assemblyTasks.ToArray());
83              Log("All taks done...");
                if (Parameters.LoadExtCntgs)
85              {
                    ExpandContigs();
87                  Log("Contigs Expanded...");
                    Log("Done!");
89              }
            }
91          else
            {
93              Log("Sequential Run");
                foreach (int k in Parameters.KList)
95              {
                    Log("Assembly for K = " + k);
97                  AssemblyCore assemblyRun = new AssemblyCore(
```

```
    k);
                    assemblyRun.DoThreadedAssembly();
                    assemblyRun = null;
                }
                if (Parameters.LoadExtCntgs)
                {
                    ExpandContigs();
                    Log("Contigs Expanded...");
                    Log("Done!");
                }
            }
        }
}
```

- AssemblyCore class containing full code for DNA assembly algorithm

```
public class AssemblyCore
{
    #region Public Global Containers

    public Dictionary<byte[], Kmer> Kmers = new Dictionary<
    byte[], Kmer>(new ByteArrayComparer());
    public Dictionary<byte[], byte[]> uu_graph = new
    Dictionary<byte[], byte[]>(new ByteArrayComparer());
    public Dictionary<byte[], byte[]> majorityVoted_graph =
    new Dictionary<byte[], byte[]>(new ByteArrayComparer());
```

```
    public Dictionary<byte[], byte[]> fu_graph = new
Dictionary<byte[], byte[]>(new ByteArrayComparer());
    public Dictionary<byte[], byte[]> uf_graph = new
Dictionary<byte[], byte[]>(new ByteArrayComparer());
    public List<UUContig> contigs = new List<UUContig>();
    public int K;
    public IOHandler IOHandler;
    private int kmersBitSize;

    private byte EncoderOccupiedCellsCount;
    private byte EncoderLastPos;

    #endregion

    private void FindKMers(Read read, int kmersBitSize)
    {
        int i = 0;
        while (i + K <= 100)
        {
            byte[] seq = new byte[kmersBitSize];
            Utilities.BitCopyArray(read.Seq, i, ref seq, 0,
K);
            Kmer kmer;
            if (!this.Kmers.ContainsKey(seq))
            {
                kmer = new Kmer();
                this.Kmers.Add(seq, kmer);
```

```
            }
            else
                kmer = this.Kmers[seq];

            if (i + K < 100)
            {
                byte rightExtensionChar = Utilities.
GetBPByIndex(read.Seq, i + K, 100);
                int rightExtensionQual = read.
GetQualityValue(i + K);
                if (rightExtensionQual >= Parameters.
HighQualityMinThreshold)
                {
                    int dnabpIndex = Utilities.GetDNABPIndex
(rightExtensionChar);
                    if (kmer.RightCounts[dnabpIndex] < 254)
//255 is reserved..
                    {
                        kmer.RightCounts[dnabpIndex]++;
                    }
                }
            }

            if (i > 0)
            {
                byte leftExtensionChar = Utilities.
GetBPByIndex(read.Seq, i - 1, 100);
```

```
53                int leftExtensionQual = read.GetQualityValue
(i - 1);
                  if (leftExtensionQual >= Parameters.
HighQualityMinThreshold)
55                {
                      int dnabpIndex = Utilities.GetDNABPIndex
(leftExtensionChar);
57                    if (kmer.LeftCounts[dnabpIndex] < 254)
//255 is reserved..
                      {
59                        kmer.LeftCounts[dnabpIndex]++;
                      }
61                }
              }
63            if (kmer.RepeatsInReads < 255)
                  kmer.RepeatsInReads++;//it is stored in
bytes not integer... so no more than 255 is possible.
65

              i++;
67        }
      }
69

      #region Public Methods
71

      public AssemblyCore(int _K)
73    {
          K = _K;
```

```
75          EncoderOccupiedCellsCount = (byte) Math.Ceiling(((
    decimal) K/4));
            EncoderLastPos = (byte) ((K%4 == 0) ? 8 : (K%4)*2);
77          //EncoderLastPos = (byte) ((K%4)*2);
            IOHandler = new IOHandler(this);
79      }

81      public void DetectKmers()
        {
83          kmersBitSize = EncoderOccupiedCellsCount;
            //Parallel.ForEach(AssemblyGlobal.Reads, read =>
    FindKMers(read, kmersBitSize));
85          foreach (Read read in AssemblyGlobal.Reads)
            {
87              FindKMers(read, kmersBitSize);
            }
89      }

91      public void DoThreadedAssembly()
        {
93          string msg;

95          DateTime t0 = DateTime.Now;
            DateTime startTime = DateTime.Now;
97

            DetectKmers();
99
```

```
        DateTime t1 = DateTime.Now;
101     AssemblyGlobal.Log(string.Format("detecting k-mers:
    {0} ms", (t1 - t0).TotalSeconds));
        AssemblyGlobal.Log("Kmers Count: " + Kmers.Count);
103
        t0 = DateTime.Now;
105     //SetStatus("Removing less frequent k-mers...");
        RemoveLessFrequentKmers(out msg);
107     //Log(msg);
        t1 = DateTime.Now;
109     AssemblyGlobal.Log(string.Format("removing less
    frequent k-mers: {0} ms", (t1 - t0).TotalSeconds));
        AssemblyGlobal.Log("Kmers Count: " + Kmers.Count);
111
        //SetStatus("Removing not receiprocal links...");
113     t0 = DateTime.Now;
        RemoveNotReceiprocalLinks(out msg);
115     //Log(msg);
        t1 = DateTime.Now;
117     AssemblyGlobal.Log(string.Format("finding extensions
    : {0} ms", (t1 - t0).TotalSeconds));
        AssemblyGlobal.Log(string.Format("uu_graph count:
    {0}", uu_graph.Count));
119     if (K == 19)
        {
121         StreamWriter sw = new StreamWriter(@"C:/
    uu_graph_cs.txt");
```

```
            foreach (KeyValuePair<byte[], byte[]> pair in
    uu_graph)
123         {
                foreach (byte b in pair.Key)
125             {
                    sw.Write(b + ",");
127             }
                sw.WriteLine(" " + pair.Value[0] + "-" +
    pair.Value[1]);
129         }
            sw.Flush();
131         sw.Close();
        }
133
        //SetStatus("Creating contigs...");
135     t0 = DateTime.Now;
        CreateContigs(out msg);
137     //Log(msg);
        t1 = DateTime.Now;
139     AssemblyGlobal.Log(string.Format("creating contigs
    based on UU graph: {0} ms", (t1 - t0).TotalSeconds));

141     PrintContigs(contigs);
        AssemblyGlobal.contigs.AddRange(this.contigs);
143     TimeSpan duration = DateTime.Now - startTime;
        AssemblyGlobal.Log("Assembly for K " + K + "
    finished!" + " (Time: " + duration.TotalSeconds + "
```

```
    Seconds)");
145  }

147  private void PrintContigs(List<UUContig> contigs)
     {
149      contigs.Sort((c1, c2) => c2.Sequence.Length.
     CompareTo(c1.Sequence.Length));
         List<UUContig> smallToBeRemoved = new List<UUContig
     >();
151      StringBuilder sb = new StringBuilder();
         int idx = 1;
153      foreach (UUContig contig in contigs)
         {
155          if (contig.OriginalSize < Parameters.
     ContigsMinLength)
             {
157              smallToBeRemoved.Add(contig);
                 continue;
159          }
             sb.AppendLine(">Contig_" + idx.ToString() + "
     _Size_" + contig.OriginalSize);
161          sb.AppendLine(Utilities.BitDecode(contig.
     Sequence, contig.OccupiedCellCounts, contig.LastPos));
             idx++;
163      }
         string directory = Path.GetDirectoryName(Parameters.
     ContigsOutputAddress);
```

```
165         StreamWriter sw = new StreamWriter(Path.Combine(
    directory, "contigs_K_" + K.ToString() + ".fa"));
            sw.Write(sb.ToString());
167         sw.Flush();
            sw.Close();
169         foreach (var uuContig in smallToBeRemoved)
            {
171             contigs.Remove(uuContig);
            }
173     }

175     public void CalculateReadsOverlaps(out string msg)
        {
177         msg = string.Empty;
            /*
179                     //finding reads overlaps (TESTING...)
                        int totalOverlaps = 0;
181                     foreach (Read read in Reads)
                        {
183                         if (read.Kmers.Count <= 1)
                                continue;
185                         Kmer lastKmer = read.Kmers.Last();
                            for (int i = 0; i < lastKmer.
    StartIndicesinReads.Count; i++)
187                         {
                                if (lastKmer.StartIndicesinReads
    [i] == 0)
```

```
189                                 {
                                        read.Overlaps.Add(lastKmer.
    Reads[i]);
191                                 }
                                }
193                                 totalOverlaps += read.Overlaps.Count
    ;
                            }
195
                            msg = "Average Overlap Count Per Read: "
    + (double) totalOverlaps/(double) Reads.Count;
197             */
    }
199
    public void CalculateReadsOverlaps()
201 {
        string msg = "";
203     CalculateReadsOverlaps(out msg);
    }
205
    public void RemoveLessFrequentKmers(out string msg)
207 {
        int min_multiplicity = Int32.MaxValue;
209     int max_multiplicity = Int32.MinValue;
        double multi_avg = 0;
211     int less_than_threshold_count = 0;
        double less_than_threshold_count_average = 0;
```

```
213         List<byte[]> toBeRemoved = new List<byte[]>();
            foreach (KeyValuePair<byte[], Kmer> kmerP in Kmers)
215         {
                Kmer kmer = kmerP.Value;
217             byte[] kmerStr = kmerP.Key;
                if (kmer.RepeatsInReads < min_multiplicity)
219                 min_multiplicity = kmer.RepeatsInReads;
                else if (kmer.RepeatsInReads > max_multiplicity)
221                 max_multiplicity = kmer.RepeatsInReads;
                if (kmer.RepeatsInReads < Parameters.
    MultiplicityMinThreshold)
223             {
                    less_than_threshold_count++;
225                 less_than_threshold_count_average += kmer.
    RepeatsInReads;
                    toBeRemoved.Add(kmerStr);
227             }
                multi_avg += kmer.RepeatsInReads;
229         }

231         multi_avg /= Kmers.Count;

233         msg =
                String.Format(
235                 "{0} kmers are less frequent! (multiplicity
    less than {1}) {2} multiplicity average : {3} {4} kmers
    count : {5}",
```

```
                toBeRemoved.Count,
237             Parameters.MultiplicityMinThreshold,
Environment.NewLine, multi_avg, Environment.NewLine,
                Kmers.Count);
239
    less_than_threshold_count_average /=
less_than_threshold_count;
241
    foreach (byte[] kmer in toBeRemoved)
    {
243     Kmers.Remove(kmer);
        /*Kmer removedOne;
245     Kmers.TryRemove(kmer, out removedOne);
        //RemovedKmers.Add(kmer, kmer);*/
    }
249 }

251 public void RemoveLessFrequentKmers()
{
253     string msg = "";
    RemoveLessFrequentKmers(out msg);
255 }

257 public void RemoveNotReceiprocalLinks(out string msg)
{
259     msg = "";
    int uuKmersCount = 0;
```

```
261         int majorityVotedCount = 0;
            int fuKmersCount = 0;
263         int ufKmersCount = 0;
            int kmerIdx = -1;
265         foreach (KeyValuePair<byte[], Kmer> kmerP in Kmers)
            {
267             kmerIdx++;
                Kmer kmer = kmerP.Value;
269             byte[] kmerStr = kmerP.Key;

271             byte rightFirstChar = (byte) 'Z';
                byte leftFirstchar = (byte) 'Z';
273             byte rightMajorityVotedChar = (byte) 'Z';
                byte leftMajorityVotedChar = (byte) 'Z';
275             bool rightExtensionIsNotUnique = false;
                bool leftExtensionIsNotUnique = false;
277
                byte[] dnaBPs = new byte[] {(byte) 'A', (byte) '
    C', (byte) 'G', (byte) 'T'};
279
                for (int i = 0; i < dnaBPs.Length; i++)
281             {
                    int bpDNAIndex = i;
283
                    //first checking left links....
285                 if (kmer.LeftCounts[bpDNAIndex] != 0)
                    {
```

```
287                     byte bpChar = dnaBPs[i];
                        //byte[] leftLinkSeq = new byte[kmerStr.
    Length];
289                     //leftLinkSeq[0] = bpChar;
                        //Array.Copy(kmerStr, 0, leftLinkSeq, 1,
     kmerStr.Length - 1);
291                     byte[] leftLinkSeq = new byte[kmerStr.
    Length];
                        Utilities.BitCopyArray(new byte[] {(byte
    ) i}, 0, ref leftLinkSeq, 0, 1);
293                     Utilities.BitCopyArray(kmerStr, 0, ref
    leftLinkSeq, 1, K - 1);
                        if (Kmers.ContainsKey(leftLinkSeq))
295                     {
                            Kmer leftLinkKmer = Kmers[
    leftLinkSeq];
297                         if (RightLinkCheck(leftLinkSeq,
    leftLinkKmer, kmerStr))
                            {
299                             if (leftFirstchar != (byte) 'Z'
    && leftFirstchar != bpChar)
                                    leftExtensionIsNotUnique =
    true;
301                             leftFirstchar = bpChar;
                                //kmer.FinalLeftCounts[
    bpDNAIndex]++;
303                         }
```

```
                    else
305                     kmer.LeftCounts[bpDNAIndex] =
byte.MaxValue;
                }
307             else
                    kmer.LeftCounts[bpDNAIndex] = byte.
MaxValue;
309         }

311         if (kmer.RightCounts[bpDNAIndex] != 0)
            {
313             byte bpChar = dnaBPs[i];
                byte[] rightLinkSeq = new byte[kmerStr.
Length];
315             Utilities.BitCopyArray(kmerStr, 1, ref
rightLinkSeq, 0, K - 1);
                Utilities.BitCopyArray(new byte[] {(byte
) i}, 0, ref rightLinkSeq, K - 1, 1);
317
                //byte[] rightLinkSeq = new byte[kmerStr
.Length];
319             //rightLinkSeq[rightLinkSeq.Length - 1]
= bpChar;
                //Array.Copy(kmerStr, 1, rightLinkSeq,
0, kmerStr.Length - 1);
321             if (Kmers.ContainsKey(rightLinkSeq))
                {
```

```
323                         Kmer rightLinkKmer = Kmers[
    rightLinkSeq];
                            if (LeftLinkCheck(rightLinkSeq,
    rightLinkKmer, kmerStr))
325                         {
                                if (rightFirstChar != (byte)'Z'
    && rightFirstChar != bpChar)
327                                 rightExtensionIsNotUnique =
    true;
                                rightFirstChar = bpChar;
329                             //kmer.FinalRightCounts[
    bpDNAIndex]++;
                            }
331                         else
                                kmer.RightCounts[bpDNAIndex] =
    byte.MaxValue;
333                     }
                        else
335                         kmer.RightCounts[bpDNAIndex] = byte.
    MaxValue;
                    }
337             }

339             /*if (kmer.LeftCounts[0] == byte.MaxValue ||
    kmer.LeftCounts[1] == byte.MaxValue || kmer.LeftCounts
    [2] == byte.MaxValue || kmer.LeftCounts[3] == byte.
    MaxValue)
```

```
                Console.WriteLine("sdfss!!!!");*/
341             /*if (kmer.FinalLeftCounts[0] + kmer.
    FinalLeftCounts[1] + kmer.FinalLeftCounts[2] + kmer.
    FinalLeftCounts[3] == 0)*/
                if (((kmer.LeftCounts[0] == byte.MaxValue ? 0 :
    kmer.LeftCounts[0]) +
343                 (kmer.LeftCounts[1] == byte.MaxValue ? 0 :
    kmer.LeftCounts[1]) +
                    (kmer.LeftCounts[2] == byte.MaxValue ? 0 :
    kmer.LeftCounts[2]) +
345                 (kmer.LeftCounts[3] == byte.MaxValue ? 0 :
    kmer.LeftCounts[3])) == 0)
                    //I'm not using sum method or for loop
    for better performance
347                 //if (kmer.FinalLeftExtensions.Count ==
    0)
                    kmer.LeftExtension = ExtensionLabel.
    DEAD_END;
349             else if (!leftExtensionIsNotUnique)
                {
351                 kmer.LeftExtension = ExtensionLabel.RESOLVED
    ;
                    //kmer.LeftExtendedChar = leftFirstchar;
353             }
                else
355             {
                    if (CheckMajorityVoting(kmer, true, out
```

```
    leftMajorityVotedChar))
357                 {
                        kmer.LeftExtension = ExtensionLabel.
    MAJORITY_VOTED;
359                     //kmer.LeftExtendedChar =
    majority_voted_char;
                    }
361                 else
                    {
363                     kmer.LeftExtension = ExtensionLabel.
    UNRESOLVED;
                    }
365             }

367             /*if (kmer.RightCounts[0] == byte.MaxValue ||
    kmer.RightCounts[1] == byte.MaxValue || kmer.RightCounts
    [2] == byte.MaxValue || kmer.RightCounts[3] == byte.
    MaxValue)
                    Console.WriteLine("sdfss!!!!");*/
369             /*if (kmer.FinalRightCounts[0] + kmer.
    FinalRightCounts[1] + kmer.FinalRightCounts[2] + kmer.
    FinalRightCounts[3] == 0)*/
                if (((kmer.RightCounts[0] == byte.MaxValue ? 0 :
     kmer.RightCounts[0]) +
371                 (kmer.RightCounts[1] == byte.MaxValue ? 0 :
    kmer.RightCounts[1]) +
                    (kmer.RightCounts[2] == byte.MaxValue ? 0 :
```

```
    kmer.RightCounts[2]) +
373                 (kmer.RightCounts[3] == byte.MaxValue ? 0 :
    kmer.RightCounts[3])) == 0)
                    //if (kmer.FinalRightExtensions.Count == 0)
375                 kmer.RightExtension = ExtensionLabel.
    DEAD_END;
                else if (!rightExtensionIsNotUnique)
377             {
                    kmer.RightExtension = ExtensionLabel.
    RESOLVED;
379                 //kmer.RightExtendedChar = rightFirstChar;
                }
381             else
                {
383
                    if (CheckMajorityVoting(kmer, false, out
    rightMajorityVotedChar))
385                 {
                        kmer.RightExtension = ExtensionLabel.
    MAJORITY_VOTED;
387                     //kmer.RightExtendedChar =
    rightMajorityVotedChar;
                    }
389                 else
                    {
391                     kmer.RightExtension = ExtensionLabel.
    UNRESOLVED;
```

```
                }
393         }

395         if (kmer.RightExtension == ExtensionLabel.
    RESOLVED && kmer.LeftExtension == ExtensionLabel.
    RESOLVED)
            {
397             uuKmersCount++;
                uu_graph.Add(kmerStr, new byte[] {
    leftFirstchar, rightFirstChar});
399         }
            else if ((kmer.LeftExtension == ExtensionLabel.
    UNRESOLVED ||
401                       kmer.LeftExtension == ExtensionLabel
    .DEAD_END) &&
                      kmer.RightExtension ==
    ExtensionLabel.RESOLVED)
403         {
                fu_graph.Add(kmerStr, new byte[] {
    rightFirstChar});
405             fuKmersCount++;
            }
407         else if ((kmer.LeftExtension == ExtensionLabel.
    UNRESOLVED ||
                      kmer.LeftExtension == ExtensionLabel
    .DEAD_END) &&
409                   kmer.RightExtension ==
```

```
ExtensionLabel.RESOLVED)
            {
411             fu_graph.Add(kmerStr, new byte[] {
leftFirstchar});
                ufKmersCount++;
413         }
            else
415         {
                //TODO: need to think more for here....
417             if (Parameters.ResolveNotUUExtensions)
                {
419                 if ((kmer.LeftExtension ==
ExtensionLabel.MAJORITY_VOTED &&
                            kmer.RightExtension ==
ExtensionLabel.RESOLVED)
421                         ||
                            (kmer.LeftExtension ==
ExtensionLabel.MAJORITY_VOTED &&
423                             kmer.RightExtension ==
ExtensionLabel.MAJORITY_VOTED)
                            ||
425                         (kmer.LeftExtension ==
ExtensionLabel.RESOLVED &&
                            kmer.RightExtension ==
ExtensionLabel.MAJORITY_VOTED))
427                 {
                        /*uuKmersCount++;
```

```
                        uu_graph.Add(kmer, String.Format("
{0}{1}", kmer.LeftExtendedChar, kmer.RightExtendedChar))
;*/
                        majorityVotedCount++;
                        majorityVoted_graph.Add(kmerStr, new
 byte[]

    {

        leftMajorityVotedChar,

        rightMajorityVotedChar

    });
                    }
                    else
                    {
                        //MessageBox.Show("sdfsd");
                    }
                }
            }
        }
    }

    public void RemoveNotReceiprocalLinks()
    {
        string msg = "";
```

```
449             RemoveNotReceiprocalLinks(out msg);
        }
451
        private bool CheckMajorityVoting(Kmer kmer, bool
    check_left_extension, out byte majorityVotedChar)
453     {
            int a_count, c_count, g_count, t_count;
455         if (check_left_extension)
            {
457             a_count = kmer.LeftCounts[0];
                c_count = kmer.LeftCounts[1];
459             g_count = kmer.LeftCounts[2];
                t_count = kmer.LeftCounts[3];
461         }
            else
463         {
                a_count = kmer.RightCounts[0];
465             c_count = kmer.RightCounts[1];
                g_count = kmer.RightCounts[2];
467             t_count = kmer.RightCounts[3];
            }
469
            int total = a_count + c_count + g_count + t_count;
471
            if ((double)a_count / (double)total >= Parameters.
    MajorityVotingThreshold)
473         {
```

```
            majorityVotedChar = (byte)'A';
475         return true;
        }
477     else if ((double)c_count / (double)total >=
    Parameters.MajorityVotingThreshold)
        {
479         majorityVotedChar = (byte)'C';
            return true;
481     }
        else if ((double)g_count / (double)total >=
    Parameters.MajorityVotingThreshold)
483     {
            majorityVotedChar = (byte)'G';
485         return true;
        }
487     else if ((double)t_count / (double)total >=
    Parameters.MajorityVotingThreshold)
        {
489         majorityVotedChar = (byte)'T';
            return true;
491     }
        majorityVotedChar = (byte)'Z';
493     return false;
    }
495

    private void RemoveKmer(byte[] originalKmer, Dictionary<
    byte[], byte[]> mapping)
```

```
497     {
            mapping.Remove(originalKmer);
499         string origDecoded = Utilities.BitDecode(
        originalKmer, this.EncoderOccupiedCellsCount, this.
        EncoderLastPos);
            string origDecodedRC = Utilities.GetRC(origDecoded);
501         byte[] rcEncoded = Utilities.BitEncode(origDecodedRC
        );
            mapping.Remove(rcEncoded);
503     }

505     public void CreateContigs(out string msg)
        {
507         int totalContigsSize = 0;
            int largestContigSize = Int32.MinValue;
509         UUContig largestContig = null;
            while (uu_graph.Count > 0)
511         {
                UUContig contig;
513             KeyValuePair<byte[], byte[]> firstElem =
        uu_graph.First();

515             RemoveKmer(firstElem.Key, uu_graph);

517             //contig.Sequence = firstElem.Key.ToList();
                int contigPos;
519             int contigCellCount;
```

```
            if (EncoderLastPos == 8)
521         {
                contigPos = 0;
523             contigCellCount = EncoderOccupiedCellsCount
    + 1;
            }
525         else
            {
527             contigPos = EncoderLastPos;
                contigCellCount = EncoderOccupiedCellsCount;
529         }
            //contig = new UUContig(firstElem.Key,
    EncoderOccupiedCellsCount, EncoderLastPos, K);
531         contig = new UUContig(firstElem.Key,
    contigCellCount, contigPos, K);
            byte leftExtension = firstElem.Value[0];
533         byte rightExtension = firstElem.Value[1];
            bool rightTruncated = false;
535         bool leftTruncated = false;

537         bool finish = false;

539         while (!finish)
            {
541             finish = true;

543             byte[] rightExtensionKmer = new byte[
```

```
kmersBitSize];
                Utilities.BitCopyArray(contig.Sequence,
contig.OriginalSize - (K - 1), ref rightExtensionKmer,
0, K - 1);
545             Utilities.BitCopyArray(new byte[] {(byte)
Utilities.GetDNABPIndex(rightExtension)}, 0, ref
rightExtensionKmer, K - 1, 1);

547             //Array.Copy(contig.Sequence.ToArray(),
contig.Sequence.Count - (K - 1), rightExtensionKmer, 0,
K - 1);
                //rightExtensionKmer[rightExtensionKmer.
Length - 1] = rightExtension;
549             //byte[] rightExtensionKmerEncoded =
Utilities.BitEncode(rightExtensionKmer);

551             /*string rightExtensionKmer =
                    contig.Sequence.Substring(contig.
Sequence.Length - (latestRightKmerSize - 1),
553
latestRightKmerSize - 1) + rightExtension;*/
                //byte[] rightExtensionKmerRC = Utilities.
GetRC(rightExtensionKmer);
555             byte[] leftExtensionKmer = new byte[
kmersBitSize];
                Utilities.BitCopyArray(new byte[] { (byte)
Utilities.GetDNABPIndex(leftExtension) }, 0, ref
```

```
    leftExtensionKmer, 0, 1);
557                 Utilities.BitCopyArray(contig.Sequence, 0,
    ref leftExtensionKmer, 1, K - 1);

559                 //byte[] leftExtensionKmer = new byte[K];
                    //leftExtensionKmer[0] = leftExtension;
561                 //Array.Copy(contig.Sequence.ToArray(), 0,
    leftExtensionKmer, 1, K - 1);
                    //byte[] leftExtensionKmerEncoded =
    Utilities.BitEncode(leftExtensionKmer);
563                 //string leftExtensionKmer = leftExtension +
     contig.Sequence.Substring(0, latestLeftKmerSize - 1);
                    //byte[] leftExtensionKmerRC = Utilities.
    GetRC(leftExtensionKmer);
565
                    /*Kmer rightExtensionKmer =
567                     new Kmer(
                            firstElem.Key.Sequence.Substring(1,
    firstElem.Key.Sequence.Length - 1) + firstElem.Value[1],
569                         string.Empty);*/
                    /*Kmer leftExtensionKmer =
571                     new Kmer(
                            firstElem.Value[0] + firstElem.Key.
    Sequence.Substring(0, firstElem.Key.Sequence.Length - 1)
    ,
573                         string.Empty);*/
```

```
575                 if (!rightTruncated)
                    {
577                     if (uu_graph.ContainsKey(
rightExtensionKmer))
                        {
579                         contig.ExpandByOne(rightExtension,
0);
                            //contig.Sequence.Add(rightExtension
);
581                         /*if (!reference_genome.Contains(
contig.Sequence))
                                MessageBox.Show("not a good
decision!");*/
583                         rightExtension = uu_graph[
rightExtensionKmer][1];
                            contig.RightExtensionChar =
rightExtension;
585                         //uu_graph.Remove(rightExtensionKmer
);
                            RemoveKmer(rightExtensionKmer,
uu_graph);
587                         finish = false;
                            //break;
589                     }
                        else if (majorityVoted_graph.ContainsKey
(rightExtensionKmer))
591                     {
```

```
                        contig.ExpandByOne(rightExtension,
    0);
593                     //contig.Sequence.Add(rightExtension
    );
                        rightExtension = majorityVoted_graph
    [rightExtensionKmer][1];
595                     contig.RightExtensionChar =
    rightExtension;
                        //RemoveKmer(rightExtensionKmer,
    majorityVoted_graph);
597                     //majorityVoted_graph.Remove(
    rightExtensionKmer);
                        finish = false;
599                     //MessageBox.Show("hoooh!");
                    }
601                 /*else if (fu_graph.ContainsKey(
    rightExtensionKmer))
                    {
603                     contig.Sequence += rightExtension;
                        rightExtension = fu_graph[
    rightExtensionKmer][0];
605                     latestRightKmerSize = Kmers[
    rightExtensionKmer].Sequence.Length;
                        //fu_graph.Remove(rightExtensionKmer
    );
607                     finish = false;
                    }
```

```
609                     else if (uf_graph.ContainsKey(
    rightExtensionKmer))
                        {
611                         //contig.Sequence +=
                            //MessageBox.Show("oh!");
613                     }*/
                        else
                        {
615                         rightTruncated = true;
                        }
                    }
619                 if (!leftTruncated)
                    {
621                     if (uu_graph.ContainsKey(
    leftExtensionKmer))
                        {
623                         contig.ExpandByOne(leftExtension, 1)
    ;
                            //contig.Sequence.Insert(0,
    leftExtension);
625                         //contig.Sequence = leftExtension +
    contig.Sequence;
                            /*if (!reference_genome.Contains(
    contig.Sequence))
627                             MessageBox.Show("not a good
    decision!");*/
                            leftExtension = uu_graph[
```

```
    leftExtensionKmer][0];
629                         contig.LeftExtensionChar =
    leftExtension;
                            //uu_graph.Remove(leftExtensionKmer)
    ;
631                         RemoveKmer(leftExtensionKmer,
    uu_graph);
                            finish = false;
633                         //break;
                        }
635                     else if (majorityVoted_graph.ContainsKey
    (leftExtensionKmer))
                        {
637                         contig.ExpandByOne(leftExtension, 1)
    ;
                            //contig.Sequence.Insert(0,
    leftExtension);
639                         //contig.Sequence = leftExtension +
    contig.Sequence;
                            leftExtension = majorityVoted_graph[
    leftExtensionKmer][0];
641                         contig.LeftExtensionChar =
    leftExtension;
                            //RemoveKmer(leftExtensionKmer,
    majorityVoted_graph);
643                         majorityVoted_graph.Remove(
    leftExtensionKmer);
```

```
                        finish = false;
                        //break;
                    }
                    /*else if (uf_graph.ContainsKey(
leftExtensionKmer))
                    {
                        contig.Sequence = leftExtension +
contig.Sequence;
                        leftExtension = uf_graph[
leftExtensionKmer][0];
                        latestLeftKmerSize = Kmers[
leftExtensionKmer].Sequence.Length;
                        //uf_graph.Remove(leftExtensionKmer)
;
                        finish = false;
                    }
                    else if (fu_graph.ContainsKey(
leftExtensionKmer))
                    {
                        //MessageBox.Show("oh!");
                    }*/
                    else
                    {
                        leftTruncated = true;
                    }
                }
```

```
665                 /*while (rightExtensionKmer.Sequence.Length
    > 10)//while is only useful when we have READ_SPLITTING
                    {
667                     if (uu_graph.ContainsKey(
    rightExtensionKmer))
                        {
669                         contig.Sequence += rightExtension;
                            rightExtension = uu_graph[
    rightExtensionKmer][1];
671                         latestRightKmerSize = kmers[
    rightExtensionKmer].Sequence.Length;
                            uu_graph.Remove(rightExtensionKmer);
673                         finish = false;
                            break;
675                     }
                        else
677                         rightExtensionKmer.Sequence =
    rightExtensionKmer.Sequence.Remove(0, 1);
                    }
679                 while (leftExtensionKmer.Sequence.Length >
    10)//while is only useful when we have READ_SPLITTING
                    {
681                     if (uu_graph.ContainsKey(
    leftExtensionKmer))
                        {
683                         contig.Sequence = leftExtension +
    contig.Sequence;
```

```
                        leftExtension = uu_graph[
    leftExtensionKmer][0];
685                     latestLeftKmerSize = kmers[
    leftExtensionKmer].Sequence.Length;
                        uu_graph.Remove(leftExtensionKmer);
687                     finish = false;
                        break;
689                 }
                    else
691                     leftExtensionKmer.Sequence =
                            leftExtensionKmer.Sequence.
    Remove(leftExtensionKmer.Sequence.Length - 1, 1);
693             }*/
                /*if (finish)
695             {
                    MessageBox.Show("finish!");
697             }*/
            }
699
            contigs.Add(contig);
701         /*if (contig.Sequence.Count > largestContigSize)
            {
703             largestContigSize = contig.Sequence.Count;
                largestContig = contig;
705         }
            totalContigsSize += contig.Sequence.Count;*/
707     }
```

```
        msg = String.Format("Average Contigs Size : {0} {1}
    Largest Contig Size : {2}",
709                             (double)totalContigsSize /
    contigs.Count, Environment.NewLine,
                                largestContigSize);
711
        //contigs.RemoveAll(contig => contig.Sequence.Length
    < 100);
713     }

715     public void CreateContigs()
        {
717         string msg = "";
            CreateContigs(out msg);
719     }

721     public void ExpandContigs(out string msg)
        {
723         msg = "";
            return;
725     }


727
        public void ExpandContigs()
729     {
            string msg = "";
731         ExpandContigs(out msg);
```

```
    }
733 #endregion

735 #region Private Methods

737 private bool RightLinkCheck(byte[] srcStr, Kmer srcKmer,
    byte[] destStr)
    {
739     for (int i = 0; i < 4; i++)
        {
741         if (srcKmer.RightCounts[i] == 0)
                continue;
743
            //byte[] rightLinkSeq = new byte[srcStr.Length];
745         //rightLinkSeq[rightLinkSeq.Length - 1] = bpChar
    ;
            //Array.Copy(srcStr, 1, rightLinkSeq, 0, srcStr.
    Length - 1);
747         byte[] rightLinkSeq = new byte[srcStr.Length];
            Utilities.BitCopyArray(srcStr, 1, ref
    rightLinkSeq, 0, K - 1);
749         Utilities.BitCopyArray(new byte[] {(byte) i}, 0,
     ref rightLinkSeq, K - 1, 1);

751         if (Utilities.UnsafeCompare(rightLinkSeq,
    destStr))
                return true;
```

```
753             /*if (rightLinkSeq.Equals(destStr,
    StringComparison.OrdinalIgnoreCase))
                    return true;*/
755         }
            return false;
757     }

759     private bool LeftLinkCheck(byte[] srcStr, Kmer srcKmer,
    byte[] destStr)
        {
761         for (int i = 0 ; i < 4; i++)
            {
763             if (srcKmer.LeftCounts[i] == 0)
                    continue;
765
                //byte[] leftLinkSeq = new byte[srcStr.Length];
767             //leftLinkSeq[0] = bpChar;
                //Array.Copy(srcStr, 0, leftLinkSeq, 1, srcStr.
    Length - 1);
769             byte[] leftLinkSeq = new byte[srcStr.Length];
                Utilities.BitCopyArray(srcStr, 0, ref
    leftLinkSeq, 1, K - 1);
771             Utilities.BitCopyArray(new byte[] {(byte) i}, 0,
     ref leftLinkSeq, 0, 1);
                if (Utilities.UnsafeCompare(leftLinkSeq, destStr
    ))
773                 return true;
```

```
            /*if (leftLinkSeq.Equals(destStr,
    StringComparison.OrdinalIgnoreCase))
775                 return true;*/


777         }
            return false;
779     }


781     #endregion
}
```

List of Figures

List of Tables